A Novel Hybrid Memory Architecture for High-Speed Packet Buffers in Network Nodes

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik
der Universität Stuttgart zur Erlangung der Würde
eines Doktor-Ingenieurs (Dr.-Ing.) genehmigte Abhandlung

vorgelegt von

Arthur Mutter

geb. in Sathmar

Hauptberichter:	Prof. em. Dr.-Ing. Dr. h.c. mult. Paul J. Kühn
1. Mitberichter:	Prof. Dr. rer. nat. Ernst W. Biersack, EURECOM (F)
2. Mitberichter:	Prof. Dr.-Ing. Andreas Kirstädter
Tag der Einreichung:	26. September 2011
Tag der mündlichen Prüfung:	26. März 2012

Institut für Kommunikationsnetze und Rechnersysteme
der Universität Stuttgart

2012

To Christine and Sarah.

Abstract

Routers are the prevalent type of network nodes in today's Internet. A router processes incoming packets and forwards them towards their destination. Core routers, i. e., routers that operate in the core of the Internet, contain up to hundreds and more ports to be able to interconnect many network segments. Temporary unbalanced traffic between the ports of a router can lead to overload situations. To minimize packet loss routers contain packet buffers to hold packets during times of congestion. To be able to provide the large buffering capacities required packet buffers are typically implemented with DRAM (Dynamic Random Access Memory).

One major problem of building high-speed packet buffers is that line rates and therewith the packet rates grow much faster than the random access time of DRAM decreases. The random access time of a memory bounds the rate of individual accesses to the memory device. At a line rate of 10 Gbps DRAM random access time was just short enough to meet the required access time. Since then, the gap between these values steadily increases. For example, on a 100 Gbps link an Ethernet frame can arrive every 6.7 ns but DRAM random access time is approx. 50 ns.

Using a hybrid memory architecture can close this gap by combining the strengths of both major memory technologies: short random access time of SRAM (Static Random Access Memory) and large capacity of DRAM. However, the architecture proposals in literature that provide a deterministic bandwidth suffer from high memory resource requirements and inefficient memory resource utilization. The main reasons for this are fragmentation, inefficient DRAM data bus utilization, and large required SRAM capacities. These properties limit scalability and increase costs and power consumption.

This thesis proposes a novel hybrid memory architecture for high-speed packet buffers that delivers deterministic bandwidth. The novelty of the architecture is that it significantly reduces the memory resources compared to related architectures from literature, while it provides the same functionality. Memory resources refer to the required SRAM and DRAM capacity and bandwidth, as well as to the DRAM data bus pin count. The feasibility of the architecture in hardware at high line rates is shown by a prototypical packet buffer implementation.

The thesis introduces at first fundamentals of packet buffering. It addresses the potential locations to place a packet buffer in a router, defines the term *packet buffer*, and introduces its basic building blocks. It also quantifies the requirements a packet buffer has to suffice and compares them to cutting-edge SRAM and DRAM devices. Then focus is set on hybrid memory architectures and the necessary metrics to evaluate these. Architecture proposals from literature are surveyed and their pros and cons are discussed.

The main objective for the design of the novel hybrid memory architecture was to reduce memory resource requirements without reducing functionality. Besides providing a deterministic bandwidth the design targets were reduction of the SRAM capacity and reduction of the DRAM resources compared to related architectures. The architecture features that are necessary to meet all targets simultaneously are derived. For example, packets are aggregated to blocks and only blocks are buffered in SRAM and DRAM. As aggregation eliminates fragmentation, this decreases the required bandwidth and capacity of SRAM and DRAM. To further significantly decrease the SRAM capacity the queues maintained in the SRAM share the SRAM dynamically.

The architecture contains a tail buffer (SRAM), a head buffer (SRAM), and a set of parallel DRAMs (or DRAM banks). The degree of parallelism can be freely chosen. The task of a high-speed packet buffer is to maintain a set of FIFO queues that hold the packets. Similar to related architectures the tail buffer holds the queue tails, the DRAMs hold the middle parts of the queues, and the head buffer holds the queue heads. A new memory management algorithm (MMA) is proposed. Further, two MMAs from literature are used in combination with this architecture. An MMA defines how blocks are distributed to the DRAMs and how blocks are transferred between SRAM and DRAM.

The typical metrics are derived to evaluate the architecture quantitatively: upper bounds for the tail buffer size and head buffer size, as well as the read latency of the system. For two MMAs these are formally proven. A detailed comparison of the metrics to those of other architectures is performed. It is shown, that the proposed architecture reduces the tail and head buffer sizes by up to 50 %. The read latency is similar or equal to that of other architectures.

Required DRAM resources are also compared to those of related architectures. These are also reduced significantly, as the proposed architecture is the only one that eliminates internal and external fragmentation and uses bank interleaving simultaneously. The first two properties reduce the DRAM bandwidth and capacity to the theoretical minimum, while the third minimizes the DRAM data bus pin count to provide the bandwidth. Finally, a dimensioning example for the proposed memory architecture is provided. This shows how to take advantage of the degrees of freedom, e. g., the degree of parallelism.

Feasibility of the architecture is shown by a prototypical implementation of a corresponding packet buffer. The prototype was described in VHDL and an FPGA development board served as platform. The implementation is presented in detail. Functional simulations and tests on the FPGA validate the correct behavior of the prototype. Place & Route results show that the prototype supports a line rate of over 10 Gbps providing 64 FIFO queues despite using an over 6 year old FPGA. The objective of the implementation was achieving full functionality. A new optimized implementation on an ASIC is estimated to support significantly more FIFO queues and a line rate of 100 Gbps and far more.

Concluding, the significant reduction of memory resources improves scalability towards higher line rates and higher number of queues compared to related architectures. Further, the reduction of memory resources also improves the energy efficiency as a packet buffer consists for the most part from memory.

Kurzfassung

Im heutigen Internet sind Router die vorherrschende Art von Vermittlungsknoten. Ein Router verarbeitet ankommende Pakete und leitet sie in Richtung ihres Ziels weiter. Core Router, d.h. Router die im Kern des Internets betrieben werden, können bis zu hunderte von Anschlüssen besitzen, um möglichst viele Netzwerksegmente miteinander zu verbinden. Kurzzeitig unausgeglichener Datenverkehr zwischen den Anschlüssen eines Routers kann zu Überlastsituationen führen. Um Paketverluste zu minimieren, besitzen Router Paketpuffer, die Pakete in Überlastsituationen puffern können. Um die hohen geforderten Pufferkapazitäten bereitstellen zu können, werden Paketpuffer typischerweise mit DRAM (Dynamic Random Access Memory) realisiert.

Ein wesentliches Problem beim Bau von Hochgeschwindigkeitspaketpuffern ist, dass Datenübertragungsraten und damit auch die Paketraten viel schneller wachsen als die wahlfreie Zugriffszeit von DRAM abnimmt. Die Zugriffszeit eines Speichers begrenzt die Rate einzelner Zugriffe auf den Speicherbaustein. Bei einer Datenübertragungsrate von 10 Gbit/s war die wahlfreie Zugriffszeit von DRAM gerade kurz genug, um die geforderte Zugriffszeit zu erfüllen. Seitdem wächst der Abstand zwischen diesen Werten stetig. Beispielsweise kann bei einer Datenübertragungsrate von 100 Gbit/s alle 6.7 ns ein Ethernet Rahmen ankommen, wohingegen die wahlfreie Zugriffszeit von DRAM jedoch ungefähr 50 ns beträgt.

Eine hybride Speicherarchitektur kann dieses Problem lösen, indem sie die Stärken der beiden wesentlichen Speichertechnologien kombiniert: die kurze wahlfreie Zugriffszeit von SRAM (Static Random Access Memory) und die große Kapazität von DRAM. Allerdings leiden die Architekturvorschläge aus der Literatur, die eine deterministische Bandbreite garantieren, an hohem Speicherressourcenbedarf und ineffizienter Speicherressourcennutzung. Die Hauptgründe hierfür sind Fragmentierung, ineffiziente DRAM-Datenbusausnutzung und große benötigte SRAM-Kapazitäten. Diese Eigenschaften limitieren die Skalierbarkeit und erhöhen die Kosten sowie den Energieverbrauch.

Diese Dissertation schlägt eine neue hybride Speicherarchitektur für Hochgeschwindigkeitspaketpuffer vor, die eine deterministische Bandbreite garantiert. Der Vorteil dieser neuen Architektur anderen gegenüber ist, dass sie bei gleich bleibender Funktionalität die benötigten Speicherressourcen wesentlich reduziert. Speicherressourcen beziehen sich hierbei auf die benötigte Kapazität und Bandbreite von SRAM und DRAM, sowie auf die Pinzahl des DRAM-Datenbusses. Die Realisierbarkeit der Architektur in Hardware bei hohen Datenraten wird mit Hilfe einer prototypischen Paketpufferimplementierung gezeigt.

Zunächst führt diese Dissertation Grundlagen zum Thema Paketpufferung ein. Sie zeigt dabei auf, wo Paketpuffer in einem Router platziert werden können, definiert den Begriff *Paketpuffer* und stellt seine elementaren Bestandteile vor. Sie quantifiziert die Anforderungen, die ein Paketpuffer erfüllen muss und vergleicht diese mit den Eigenschaften aktueller SRAM und DRAM-Bausteine. Anschließend wird der Fokus auf hybride Speicherarchitekturen und die notwendigen Metriken zu deren Bewertung gesetzt. Architekturvorschläge aus der Literatur werden vorgestellt, diskutiert und bewertet.

Die Zielsetzung beim Entwurf der neuen hybriden Speicherarchitektur war, die benötigten Speicherressourcen zu reduzieren ohne dabei die Funktionalität einzuschränken. Neben der Bereitstellung einer deterministischen Bandbreite waren die Entwurfsziele die Reduktion der SRAM-Kapazität und die Reduktion der DRAM-Ressourcen im Vergleich zu anderen Architekturen. Die notwendigen Architekturmerkmale zur gleichzeitigen Erfüllung aller Entwurfsziele werden hergeleitet. Zum Beispiel werden Pakete zu Blöcken aggregiert und in SRAM und DRAM werden nur Blöcke gepuffert. Da die Aggregation die Fragmentierung eliminiert, werden hierdurch die benötigte Bandbreite und die Kapazität von SRAM und DRAM gesenkt. Um die SRAM-Kapazität weiter deutlich zu verkleinern, teilen sich die Warteschlangen im SRAM den Speicherplatz dynamisch.

Die Architektur enthält einen Eingangspuffer (SRAM), einen Ausgangspuffer (SRAM) und einen Satz paralleler DRAMs (oder DRAM-Bänke). Der Grad der Parallelität ist völlig frei wählbar. Die Aufgabe eines Hochgeschwindigkeitspaketpuffers ist es, einen Satz von FIFO-Warteschlangen zu verwalten, welche die Pakete enthalten. Wie bei anderen Architekturen auch enthält der Eingangspuffer die Enden der Warteschlangen, die DRAMs enthalten die Mittelteile der Warteschlangen und der Ausgangspuffer enthält die Anfänge der Warteschlangen. Ein neuer Speicherverwaltungsalgorithmus wird vorgeschlagen. Zudem werden auch zwei aus der Literatur bekannte Speicherverwaltungsalgorithmen eingesetzt. Ein Speicherverwaltungsalgorithmus definiert, wie die Blöcke auf die DRAMs verteilt werden und wie die Blöcke zwischen SRAM und DRAM übertragen werden.

Um die Architektur quantitativ bewerten zu können, werden die typischen Metriken abgeleitet: die Obergrenze für die Größe des Eingangspuffers und des Ausgangspuffers, sowie die Leselatenz des Systems. Für zwei Speicherverwaltungsalgorithmen werden diese formal bewiesen. Ein detaillierter Vergleich der Metriken mit denen anderer Architekturen wird durchgeführt. Es wird gezeigt, dass die vorgeschlagene Architektur die Größe von Eingangspuffer und Ausgangspuffer um bis zu 50 % reduziert. Die Leselatenz ist ähnlich oder gleich zu der Leselatenz anderer Architekturen.

Auch die notwendigen DRAM-Ressourcen werden mit denen anderer Architekturen verglichen. Diese werden ebenfalls signifikant reduziert, da die vorgeschlagene Architektur die einzige ist, die jegliche interne und externe Fragmentierung eliminiert und gleichzeitig verschränkt auf DRAM-Bänke zugreift. Die ersten beiden Eigenschaften reduzieren die Bandbreite und Kapazität des DRAMs auf das theoretische Minimum, während die dritte Eigenschaft die Pinzahl des DRAM-Datenbusses minimiert, die notwendig ist, um die Bandbreite bereit zu stellen. Schließlich wird beispielhaft eine Dimensionierung für die vorgeschlagene Architektur durchgeführt. Diese zeigt wie die Freiheitsgrade vorteilhaft genutzt werden können, z. B. der frei wählbare Grad der Parallelität.

Die Realisierbarkeit der Architektur wird durch die prototypische Implementierung eines entsprechenden Paketpuffers gezeigt. Der Prototyp wurde in VHDL beschrieben und auf einer FPGA-basierten Plattform betrieben. Die Implementierung wird im Detail vorgestellt. Funktionale Simulationen und Tests auf dem FPGA validieren das korrekte Verhalten des Prototyps. Place & Route Ergebnisse zeigen, dass der Prototyp trotz der Verwendung eines 6 Jahre alten FPGAs eine Datenübertragungsrate von 10 Gbit/s und 64 FIFO-Warteschlangen unterstützt. Die Zielsetzung der Implementierung war ein voll funktionsfähiger Prototyp. Es wird abgeschätzt, dass eine neue, optimierte Implementierung auf einem ASIC eine wesentlich höhere Zahl an FIFO-Warteschlangen sowie Datenübertragungsraten von 100 Gbit/s und weit mehr unterstützen kann.

Abschließend lässt sich sagen, dass die wesentliche Reduktion der Speicherressourcen die Skalierbarkeit verbessert im Hinblick auf höhere Datenübertragungsraten und eine höhere Zahl an FIFO-Warteschlangen. Des Weiteren senkt die Reduktion der Speicherressourcen auch den Energieverbrauch, da ein Paketpuffer zum größten Teil aus Speichern besteht.

Contents

List of Figures

List of Tables

Abbreviations and Symbols

Abbreviations

ALM	Adaptive Logic Module
ALUT	Adaptive LUT
ASIC	Application Specific Integrated Circuit
CFDS	Conflict-Free DRAM System
CIO	Common I/O
CIOQ	Combined Input/Output-Queued
CMOS	Complementary Metal Oxide Semiconductor
CoS	Class of Service
CPU	Central Processing Unit
DDR	Double Data Rate
DIMM	Dual Inline Memory Module
DMM	Dynamic Memory Manager
DRAM	Dynamic Random Access Memory
DSL	Digital Subscriber Line
DSS	DRAM Scheduling Subsystem
ECQF	Earliest Critical Queue First
FBDIMM	Fully Buffered DIMM
FF	Flip-Flop
FIFO	First-In First-Out
FPGA	Field Programmable Gate Array

GDDR	Graphics Double Data Rate
HDL	Hardware Description Language
head-MMA	Head Memory Management Algorithm
HOL	Head Of Line
HSD	Hybrid SRAM/DRAM System
IOM	In-Order Matching
IP	Internet Protocol
IQ	Input-Queued
ISO	International Organization for Standardization
ISP	Internet Service Provider
JEDEC	Joint Electron Device Engineering Council
LAN	Local Area Network
LBRB	Last Block Request Buffer
LIFO	Last-In First-Out
LUT	Lookup Table
MAC	Medium Access Control
MAN	Metropolitan Area Network
MaRBD	Maximize Request Buffer Delay
MDQF	Most Deficit Queue First
MiRBD	Minimize Request Buffer Delay
MMA	Memory Management Algorithm
NP	Network Processor
OQ	Output Queue *or* Output-Queued
OSI	Open System Interconnection
PCB	Printed Circuit Board
PHSD	Parallel Hybrid SRAM/DRAM System
PLD	Programmable Logic Devices
PSM	Parallel Shared Memory

QDR	Quad Data Rate
QoS	Quality Of Service
RAM	Random Access Memory
RLDRAM	Reduced Latency DRAM
RMC	Route and Management Controller
RPP	Read Pipeline
RTT	Round Trip Time
SDR	Single Data Rate
SDRAM	Synchronous DRAM
SGRAM	Synchronous Graphics Random Access Memory
SIM	Scheduling Information Manager
SIMM	Single Inline Memory Module
SIO	Separate I/O
SLA	Service Level Agreement
SPHSD	Semi-Parallel Hybrid SRAM/DRAM System
SPMT	Serial Port Memory Technology
SPP	Silicon Packet Processor
SRAM	Static Random Access Memory
SSRAM	Synchronous SRAM
tail-MMA	Tail Memory Management Algorithm
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver/Transmitter
UHP	Universal Hardware Platform
VHDL	Very High Speed Integrated Circuit HDL
VLAN	Virtual Local Area Network
VOQ	Virtual Output Queue
WAN	Wide Area Network
WPP	Write Pipeline

Symbols

B	Block size in a system that does not incorporate parallel subsystems
b	Block size in a system that incorporates parallel subsystems, DRAMs, or DRAM banks
C	DRAM capacity of the packet buffer
f	System clock frequency
f_{min}	Minimal system clock frequency
G	Number of bank groups in the CFDS
G_{min}	Minimal inter-framing gap
k	Degree of parallelism, i. e., the number of parallel subsystems or DRAMs (or DRAM banks)
k_{min}	Minimal number of parallel subsystems in the PHSD
L	Read latency
L_{SPHSD}	Read latency of the SPHSD
$L_{HSD-ECQF}$	Read latency of the HSD when the ECQF MMA is used
$L_{CFDS-ECQF-M8}$	Read latency of the CFDS when the ECQF MMA and a DRAM with 8 banks are used
$L_{CFDS-ECQF}$	Read latency of the CFDS when the ECQF MMA is used
$L_{PHSD-max}$	Maximal read latency of the PHSD
M	Number of banks of DRAM device
N	Number of line cards
p	DRAM data bus pin count
p_{cap}	DRAM data bus pin count to achieve the required capacity
p_{bw}	DRAM data bus pin count to achieve the required bandwidth
P	Packet size *or* Number of physical flow queues maintained by the CFDS
P_{min}	Minimal packet size
P_{wc}	Worst-case packet size
Q	Number of flow queues maintained by the packet buffer
R	Aggregate line rate of all ports of a line card

R_{gross}	Gross data rate of the input bus to the packet buffer
r	Packet rate
r_{max}	Maximal packet rate
r_{wc}	Worst-case packet rate
S_{head}	Head buffer size
$S_{head-SPHSD-MiRBD}$	Head buffer size of the SPHSD when the MiRBD MMA is used
$S_{head-SPHSD-MaRBD}$	Head buffer size of the SPHSD when the MaRBD MMA is used
$S_{head-HSD-ECQF}$	Head buffer size of the HSD when the ECQF MMA is used
$S_{head-HSD-MDQF}$	Head buffer size of the HSD when the MDQF MMA is used
$S_{head-CFDS-ECQF}$	Head buffer size of the CFDS when the ECQF MMA is used
S_{tail}	Tail buffer size
$S_{tail-SPHSD}$	Tail buffer size of the SPHSD
$S_{tail-SPHSD-k1}$	Tail buffer size of the SPHSD when $k = 1$
$S_{tail-SPHSD-k16}$	Tail buffer size of the SPHSD when $k = 16$
$S_{tail-SPHSD-min}$	Tail buffer size of the SPHSD, smallest possible value
$S_{tail-HSD}$	Tail buffer size of the HSD
$S_{tail-CFDS}$	Tail buffer size of the CFDS
$S_{tail-PHSD}$	Tail buffer size of the PHSD
T_{FAW}	Four bank activation window of a DRAM device
T_{RC}	Row cycle time of a DRAM device
T_{RRD}	Row to row activation delay of a DRAM device
T	Access time of a memory device; when not denoted differently, it refers to the random access time of the memory device, i. e., the worst case access time
T_{imag}	Imagined access time in the CFDS
t_{ck}	Clock cycle time
w	Internal bus width

ρ	DRAM data bus utilization
$\lceil\ \rceil$	Round up to the next integer number

1 Introduction

1.1 Motivation

Today, communication via the Internet is indispensable for the commercial and for the private sector. This extensive use is reflected in the ever growing data rates used in the Internet. For example Ethernet line rates increased by a factor of 10 per 5 years during the past 15 years [9]. The data exchanged by the communication partners is today mainly transported by packet switching networks. The network nodes (e. g. routers) operated in these packet switching networks have to constantly provide very high throughput and high reliability. This is true especially for core networks which form the backbone of the Internet. The malfunction of a core node or just a drop of its performance would lead to large economic loss for both, user and provider.

Routers are the prevalent type of network nodes in core networks. Core routers contain up to hundreds and more ports to be able to interconnect many network segments. They process incoming packets and forward them towards their destination. Temporary unbalanced traffic between the ports of a router can lead to overload situations. To minimize packet loss routers contain packet buffers to hold packets during times of congestion.

Favorably, these packet buffers provide a deterministic bandwidth. Deterministic behavior has three main advantages over statistical: Firstly, the packet buffer delivers 100 % bandwidth under any traffic condition. Secondly, this enables implementation of routers that are safe against adversarial attacks and ”bake-off” tests as there are no loopholes an adversary (e. g., a virus or a hacker) could exploit, e. g., there is no traffic pattern that brings down the router. Thirdly, a simple and deterministic packet buffer interface leads to simple interfacing that reduces the overall complexity.

To build a packet buffer, one can choose between two prevalent memory technologies: Static Random Access Memory (SRAM) and Dynamic Random Access Memory (DRAM). SRAM has a short access time, a small capacity, and a high price per bit. DRAM has a long access time, a large capacity, and a low price per bit. To achieve low packet loss probabilities and for the Transmission Control Protocol (TCP) to perform well packet buffers require large capacities. Due to this reason packet buffers typically use DRAM.

One major problem of building high-speed packet buffers is that line rates and therewith the packet rates grow much faster than the random access time of DRAM decreases. The random access time bounds the rate of individual accesses to the memory device. The increase in packet rate is equal to the increase in line rate, since the minimal packet sizes do not change over time,

e. g., the minimal size of an Ethernet frame is 64 byte. DRAM random access time decreases by just 7 % per year [12], i. e., by 30 % per 5 years. At a line rate of 10 Gbps DRAM random access time was just short enough to meet the required access time. Since then, the gap between these values steadily increases. For example, the first commercial core routers that support 100 Gbps Ethernet appear at the time of writing. On a 100 Gbps link an Ethernet frame can arrive every 6.7 ns (calculation includes inter frame gap) but DRAM random access time is approx. 50 ns.

Using a hybrid memory architecture can close this gap. Hybrid refers to the fact that it utilizes memory of two different memory technologies to store the packet data: SRAM and DRAM. This architecture combines the strengths of both: short access time of SRAM *and* large capacity of DRAM. However, the architecture proposals in literature that provide a deterministic bandwidth suffer from high memory resource requirements. The main reasons therefore are fragmentation, inefficient DRAM data bus utilization, and large required SRAM capacities. These properties limit scalability and increase costs and power consumption.

1.2 Thesis Contribution

In this thesis the author proposes a novel hybrid memory architecture for high-speed packet buffers that deliver deterministic bandwidth. The architecture consists of three parts: a tail buffer and a head buffer made of SRAM and a set of parallel DRAM banks for bulk storage.

The architecture significantly reduces the required memory resources compared to related architectures from literature. Memory resources refer to the required capacity and bandwidth of SRAM and DRAM, as well as to the DRAM data bus pin count. Reduction is achieved by two measures: (i) reduction of memory resource requirements by architectural changes and (ii) efficient utilization of memory resources. As an example for (i), utilization of dynamic memory allocation in the tail buffer and the presence of parallel DRAM banks reduce the tail buffer size by up to 50 %. As an example for (ii), DRAM resources (capacity and data bus pin count) are significantly reduced by elimination of any fragmentation and by utilization of bank interleaving. These reductions significantly improve scalability towards higher line rates and higher number of queues.

The required tail and head buffer sizes required as well as the read latency of the system are formally proven and assessed in detail. To show that the architecture is operational and feasible the author implemented a corresponding packet buffer prototype. The functionality of the prototype was tested on an FPGA-based platform. The thesis provides detailed implementation results including block diagrams of the design, achieved line rates, and hardware resource requirements.

The author presented most of the results of this thesis on an international conference on router architectures [1]. Additionally, the author discussed with other scientists the challenges and perspectives of packet buffer design and packet processing at future line rates in a position paper [2]. The author participated in the following joint publications to related topics: as main author [3, 4], as co-author [5, 6]. Prototype implementation has been party realized by student research projects supervised by the author [7, 8].

1.3 Thesis Organization

This thesis is structured into a Chapter on fundamentals and related work (Chapter 2), a Chapter on the hybrid memory architecture proposal and its assessment (Chapter 3), and a Chapter on the prototypical implementation of an according packet buffer and its validation (Chapter 4).

Chapter 2 introduces necessary fundamentals and surveys related hybrid memory architectures from literature. It first motivates the importance of routers in the Internet today and shows the potential locations to place a packet buffer in a router. Then it defines the term *packet buffer* and introduces its basic building blocks. It also quantifies the requirements a packet buffer has to suffice. Then it introduces the prevalent memory technologies available to implement packet buffers. In the following discussion it points out that a single memory technology cannot suffice all requirements of a high-speed packet buffer simultaneously. Then it presents basic architecture approaches to overcome the limitations of the individual memory technologies. It concludes that hybrid approaches, which combine several memory technologies, are most promising. Then it focuses on hybrid memory architectures and introduces the necessary metrics to evaluate these. Finally, it presents explicit architecture proposals from literature and discusses their pros and cons.

Chapter 3 introduces and assesses the hybrid memory architecture proposal for high-speed packet buffers. First, it gives an overview of the design objectives and derives for each the required architectural features that enable it. Then it introduces in detail the hybrid memory architecture proposal, formally proves its resource requirements and properties, and compares the results to that of related architectures. A dimensioning example for the proposed architecture closes this Chapter.

Chapter 4 shows the feasibility of the hybrid memory architecture proposal by presenting the prototypical implementation of a corresponding packet buffer. First, it motivates the implemented input buffer scenario. Then it introduces the hardware platform used for implementation and the prototype properties. The core of the Chapter presents the implementation in detail. Finally, it presents the prototype's validation results with respect to functionality, hardware resource requirements, and throughput.

2 Packet Buffers for Network Nodes

The Internet is a packet switching network connecting hundreds of millions of end hosts. The hosts communicate by sending packets. The Internet consists of network nodes that route and switch the packets from source to destination end host. A network node may have up to hundreds and more ports. Temporary unbalanced traffic or traffic overload in these nodes requires high-speed packet buffers to hold packets during times of congestion.

This Chapter introduces why it is challenging to build packet buffers for high-speed network nodes and discusses the architectures available in literature that try to solve the problem. The Chapter is organized in five Sections.

Section 2.1 first introduces and classifies today's prevalent types of network nodes. Then it presents the possibilities for locating buffers in network nodes. Further, it describes exemplarily the individual building blocks of a very common high-speed network node architecture: the Combined Input/Output-Queued Router.

Section 2.2 defines the term *packet buffer* and introduces its basic building blocks. Further it presents the degrees of freedom organizing the associated memory and finally points out the explicit requirements a packet buffer has to meet.

Section 2.3 introduces the prevalent memory technologies available to implement packet buffers. In the final discussion it points out that no single memory device meets all requirements of a high-speed packet buffer simultaneously.

Section 2.4 presents the basic architecture approaches to overcome the limitations of the individual memory technologies. Finally, it concludes that hybrid approaches which combine several memory technologies are most promising.

Section 2.5 focuses on hybrid memory architectures. Thereby it first introduces necessary metrics to evaluate these architectures. Following up it presents in detail explicit architecture proposals from literature and discusses their pros and cons.

2.1 Network Nodes

Network nodes for packet switched networks have two or more ports and process each packet individually. They also receive and transmit packets individually via their input and output ports, respectively. The operations carried out on the packets depends on the node's type and ranges from determining the outgoing port (or ports) to packet modifications, deep packet inspection and queuing. Network nodes operate on different layers of the Open System Interconnection (OSI)[1] reference model. The various functionalities necessary therefore define the specific types of network nodes. This Section first gives an overview of network node types before focusing on routers and their architecture.

2.1.1 Types of Network Nodes

The four main types of network nodes are repeaters, bridges, routers and gateways [14].

Repeaters or network hubs connect network segments at the physical layer (layer 1) of the OSI model. They just broadcast the signal to adjacent network segments and have no logic functions.

A **bridge** connects multiple network segments at the data link layer (layer 2). The term switch or layer 2 switch is used interchangeably with bridge. Switches are designed to operate in a Local Area Network (LAN).

A popular example for this type of devices is an Ethernet switch. It makes forwarding decisions on the basis of Medium Access Control (MAC) addresses and without topology information. Switches use MAC-learning to keep their forwarding database up to date. Due to the flat MAC address space switches use flooding to reach unknown devices, i. e., if the switch does not find the destination MAC address of a packet in its forwarding database it sends the packet out on all switch ports except the incoming port of the packet.

A **router** connects network segments at the internetwork layer (layer 3) and was initially designed to operate in Metropolitan Area Networks (MAN) and Wide Area Networks (WAN). Today, most home networks also contain a router. Routers operate on globally unique addresses in a hierarchical address space. Further, it is itself explicitly addressable. The most prominent example is an Internet Protocol (IP) backbone router used in the Internet. A router's main task is to forward each packet towards its destination. It accesses its forwarding table to determine a packets next hop. To build its forwarding table the router uses topology information and applies a routing algorithm, e. g., "shortest path".

Finally a **gateway** connects network segments at the transport layer (layer 4) or higher. It typically operates between different network types and therefore usually transforms protocol parameters and addresses.

A core network or backbone network is the central part of a communication network. The core network of an Internet Service Provider (ISP), e. g., Deutsche Telekom, is a WAN. Routers are the predominant node type in core networks where they operate at highest data rates. Extreme data rates and high number of ports make packet buffering very challenging in these devices.

[1]The Open System Interconnection Reference Model (OSI Reference Model, [13]) is an abstract description for layered communications and network architecture. It was standardized by the International Organization for Standardization (ISO). The OSI Network Architecture defines 7 principal layers. These are from bottom to top: Physical layer (1), Data Link Layer (2), Network Layer (3), Transport Layer (4), Session Layer (5), Presentation Layer (6), and Application Layer (7).

Due to the high importance of routers and the big challenges regarding their packet buffering this thesis focuses on routers.

2.1.2 Types of Routers

Routers can be operated at different points in a network. Depending on the place of installation a router has to support different functions and processes the packets at different rates. The complexity of a router depends on these properties. Routers can be classified in many types. In this thesis we follow the proposal in [15], where the authors distinguish between three main router types: core routers, edge routers and enterprise routers.

Core routers are operated by service providers to interconnect a large number of smaller networks. The core network is also called the ”Internet-backbone” as it serves as the backbone of today’s Internet. The traffic arriving at a core router is highly aggregated and of very high speed, e. g., 10 Gbps and more per port. The core nodes are the most critical nodes in a network and therefore should not fail. ”The primary requirements for a core router are high speed and reliability” [15].

Edge routers, also known as **access routers**, are installed at the service providers’ network edge and provide connectivity to customers. One of their main tasks is aggregating the traffic from the customers and its forwarding to the core. Therefore edge routers need to support a large number of ports and a variety of access technologies, e. g., Digital Subscriber Line (DSL) or cable modems.

Enterprise routers are operated by companies, universities etc. to interconnect their end systems. Typical enterprise networks use Ethernet and are built up by inexpensive Ethernet devices like hubs and switches. Besides basic connectivity with a large number of ports these routers need to support further features like firewalls, filters and Virtual Local Area Networks (VLAN, cf. Ethernet VLAN extension of [16]). Enterprises consider the network as operations expense and try to minimize it. So routers targeting this market segment need to have a low per port cost.

This thesis targets the challenges to build high-speed packet buffers like needed in core routers. The following Sections therefore focus on these core routers.

2.1.3 Router Architectures

A core router comprises of a number of line cards that are interconnected by a backplane. A line card itself contains one or more ports.

A physical port logically divides into an input (receive) and an ouptput (transmit) part. Similarly, a physical line card also divides logically into an input (ingress) part and output (egress) part. This thesis references them as *input port*, *output port*, and as *input line card* and *output line card*, respectively. The words input and ingress as well as output and egress are used interchangeably. Depending on the implementation the two logical parts (input and output) may of course share resources.

The input line card classifies incoming packets and then processes them based on the classification result. Afterwards, the backplane transfers the packets to the output line card where they can exit on the respective output ports.

Routers can be classified with respect to many criteria. Targeting packet buffering there are two relevant criteria. The first is the *backplane type* of the router as the backplane may contain a buffer itself and in many architectures it receives the packets from the packet buffer. The second is the *buffer's location* in the router architecture. Both criteria fundamentally impact the performance and complexity of a router. The following two Sections show the classification with respect to these criteria.

The most common protocols like the IP allow the use of variable-length packets. However, backplanes and packet buffers usually operate on constant-size data elements often called *cells*. The reason for this is that these components are usually implemented as digital circuits in hardware to achieve high performance and to be able to guarantee high timing requirements in the range of nanoseconds.

As hardware design is much more complex than software design constant sized data elements help to lower complexity significantly. Therefore the input line card segments incoming variable-length packets to constant-size cells. On the output line card the packets are reassembled before they depart on the output port.

2.1.3.1 Backplane

There are two general types of backplanes [15]:

- Shared backplanes
- Switched backplanes

In case of a shared backplane, e. g., a shared bus, only two line cards can communicate at any instant. In case of a switched backplane, also referred to as switch fabric, multiple line cards can communicate simultaneously. The aggregate line rate of the interfaces on all line cards[2] defines the required bandwidth[3] of the backplane. Obviously it is much simpler to meet the throughput requirements with a switch fabric. Today all faster routers use switch fabrics.

Two basic types of switch fabrics exist [10]:

- Single-stage switch fabrics
- Multiple-stage switch fabrics

[2]Depending on the application, a line card may be also connected to the backplane with less than the aggregate line rate of its ports.

[3]In computer networking and computer science the term *bandwidth* is a *bit rate* measure for communication resources.

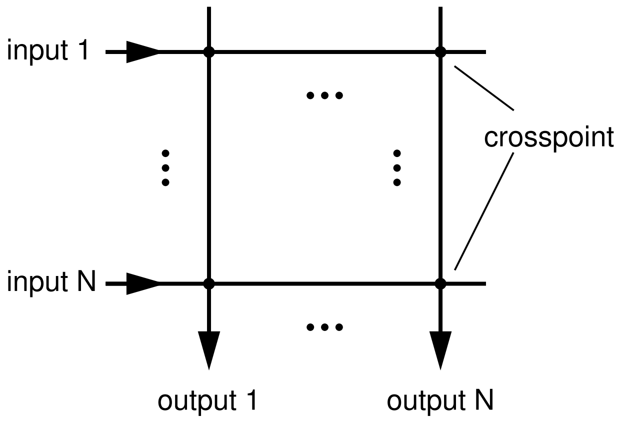


Figure 2.1: Bufferless crossbar (Source [10])

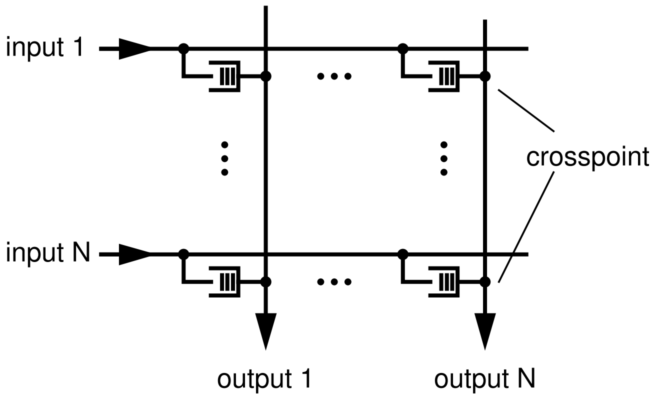


Figure 2.2: Buffered crossbar (Source [10])

A single-stage switch fabric is a fully interconnected network, where every input can be connected to every output directly. A crossbar is an example for a single-stage switch fabric (cf. Figure 2.1). The drawback of single-stage fabrics is the required central scheduler, which is challenging for a large number of inputs. The number of inputs the arbitration algorithm has to consider is the number of line cards (N) times the number of queues (Q) maintained per line card, where the latter can be quite large. Nevertheless, crossbar switch fabrics are very popular for fabric implementation due to their non-blocking capability, simplicity, modularity, and their market availability [10].

A multiple-stage switch fabric is a network of switch modules [10]. Each module is a small switch with two ore more ports. Here each input is connected to a switch module and not directly to the outputs. To reach the output line card every packet traverses several switching modules. Examples for multiple-stage switching networks are Clos [17] and Banyan [18] networks.

Switch fabrics can be further classified into [10]:

- Bufferless switch fabrics
- Buffered switch fabrics

The drawback of bufferless switch fabrics is the need for a centralized scheduler. This can be solved by using a buffered crossbar switch fabric with buffers at each crosspoint as these make scheduling distributed. Figure 2.2 shows a buffered crossbar. A buffered crossbar scheduler consists of N input schedulers and N output schedulers which work independently and parallel [11].

The number of cross points is $N \times N$. For large N, like in the order of 256 and higher, the total required buffer memory can be quite large. However, currently crossbar switches are limited by the number of I/O pins required for data transfer to and from the chip [19]. This leaves empty area on the die to implement memory. The drawback of buffered crossbars is that they suffer from so called "crosspoint blocking" what occurs, when a low priority cell resides in a crosspoint buffer. This prevents a higher priority cell of the same input/output pair to enter the crossbar as the corresponding buffer is occupied.

In multiple-stage fabrics buffering helps to resolve packet contention during switching but also may cause a sequencing problem. They therefore require output line cards capable of reordering cells when necessary.

The electrical implementation of switch fabrics is limited towards large number of line cards and high data rates due to increasing power and chip count requirements. Here hybrid approaches with electrical scheduling and optical switching are considered to be more scalable, but are yet unsolved [10].

2.1.3.2 Buffer Location

In a router or switch it can easily happen that an output port is temporarily overbooked because packets from several input ports are destined to it in a short time period. These packets now compete for the output port resource. During classification on the input line card each packet is assigned a Class of Service (CoS). Later, a scheduler uses this CoS information to define the packet transmission order on each output port.

To reduce packet drops routers buffer these packets for the duration of the congestion. The buffer location essentially influences the router performance and implementation complexity.

Possibilities for the buffer location are on the input line card, the output line card, or on input and output line card. This leads to the three main queuing variants known in literature:

- Input-Queued (IQ)
- Output-Queued (OQ)
- Combined Input/Output-Queued (CIOQ)

For simplification of description it is assumed that each line card has only one port with line rate R. In reality a line card may contain several ports with an aggregate line rate of R.

The content of the following descriptions and statements about IQ, OQ and CIOQ routers mainly originate from [11].

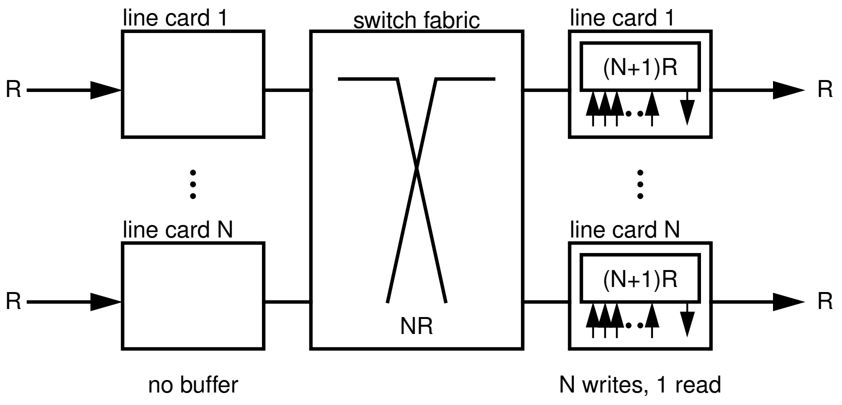


Figure 2.3: Output-Queued Router with distributed memories (Source [11])

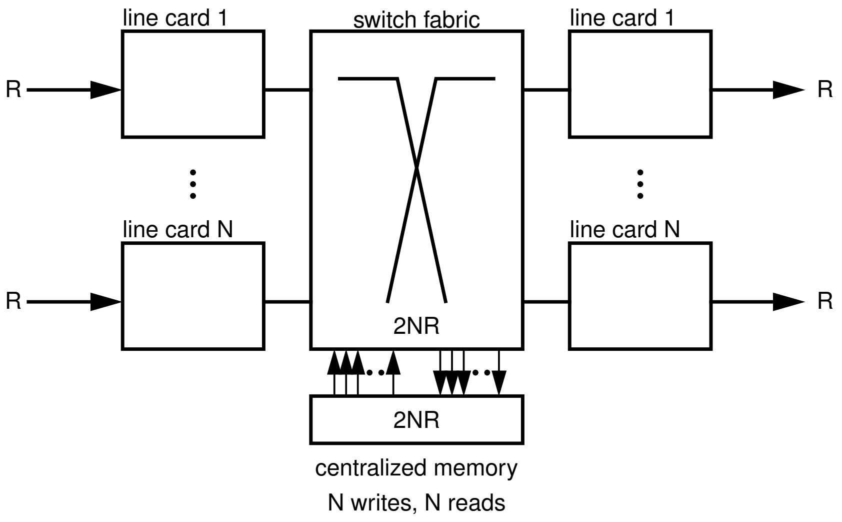


Figure 2.4: Output-Queued Router with centralized shared memory (Source [11])

OQ Router

An **OQ router** has a buffer at each output (cf. Figure 2.3). The input line card processes and segments a packet to constant-size cells. The switch fabric transfers the cells to the corresponding output line card. In an *NxN* OQ router each output buffer has to be capable to write *N* cells and read one cell simultaneously. With this, the memory bandwidth of every individual memory is $(N+1)R$. The required bandwidth of the switch fabric is NR.

Another variant of an OQ router is the centralized shared memory router (Figure 2.4). It uses one central memory that is shared by all line cards instead of individual memories on the line cards. This central memory requires a bandwidth of $2NR$ and a switch fabric bandwidth of $2NR$. The big advantage of the centralized memory is that its capacity can be shared by all data flows from all line cards, leading to a smaller total required capacity. However, when memories are distributed like in the first variant, then memory can be added incrementally with new line cards.

Due to the high memory bandwidth required, OQ routers are not scalable even for moderate line rates. However, OQ routers perform like the ideal router as there is no fabric contention and the

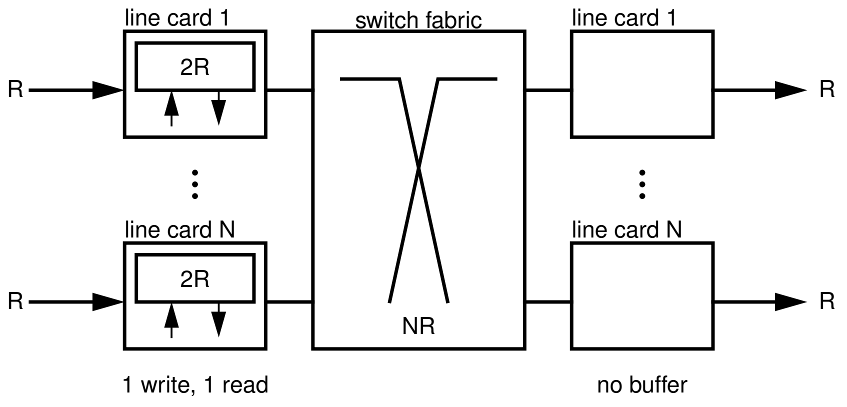


Figure 2.5: Input-Queued Router (Source [11])

scheduler on the output line card can perform an optimal scheduling. Therefore OQ routers are often used for benchmarking and comparison.

IQ Router

An **IQ router** buffers incoming packets on the input line card (cf. Figure 2.5). Therefore a memory needs to perform only one write and one read operation simultaneously, what is the minimum for a buffer. This leads to a memory bandwidth of $2R$. The switch fabric has a bandwidth of NR. However, memories could be dimensioned to have a higher read bandwidth, but this is not mandatory.

If the buffered cells are organized as one First-In First-Out (FIFO) queue, only the Head Of Line (HOL) cell is considered for switching. When a HOL cell is not processed due to fabric contention, every cell behind it is blocked even if its destined output is currently idle. This phenomenon is called the HOL blocking and limits router throughput significantly [20].

HOL blocking can be completely eliminated by the use of Virtual Output Queues (VOQ). Here, each input buffer maintains one FIFO queue per output line card and per CoS. So on a line card no cell destined to an output can block a cell destined to a different output. The contention resolution algorithm considers the total number of $N^2 \times$ *number of CoS* VOQs of all line cards. IQ routers have to resolve fabric contention before cells are forwarded to the output line cards. Contention resolution works by means of a matching process. The matching process determines pairs of switch fabric input and switch fabric output (input i, output j). After matching the switch fabric is configured and the cells are transferred simultaneously from inputs to outputs. Complex maximum weight matching algorithms that provide 100 % throughput are not practicable at high speeds [11]. Therefore, most routers use heuristic algorithms like iSLIP [21] that are often realized iteratively.

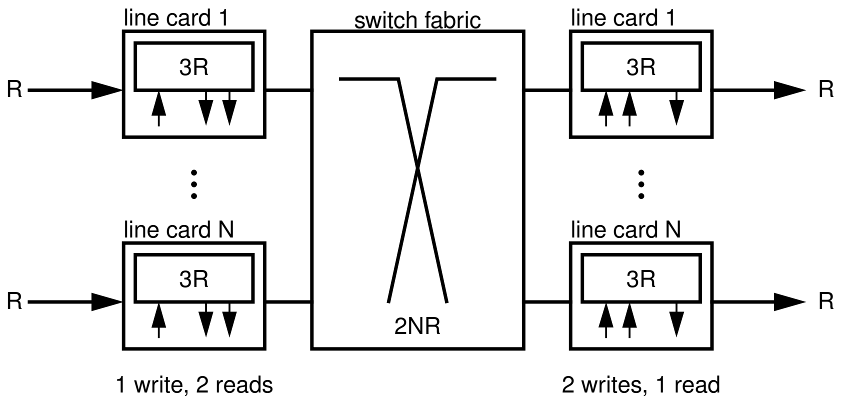


Figure 2.6: Combined Input/Output-Queued Router (Source [11])

CIOQ Router

IQ and OQ routers are often also referred to as single buffered routers as they have only a single stage of buffering. In contrast to these, a **CIOQ router** buffers packets on the input as well as on the output line cards (cf. Figure 2.6).

The central switch fabric arbitration becomes easier with two stages of buffering [11]. This is due to the fact, that the input line card can buffer a cell when the switch fabric is not free. Later, when the output port is busy while forwarding the cell via the switch fabric, the output line card can buffer the cell until the output port is idle.

According to [11], if the memory on each line card has a bandwidth of $3R$ and the router implements a time reservation scheduling algorithm this router can behave like an OQ router that supports Quality Of Service (QoS). Therefore the switch fabric needs a bandwidth of $2NR$. Since the switch fabric needs to operate faster than in an IQ router, this shortens the time available for the matching process [10]. Therefore, CIOQ routers with unbuffered crossbar are hard to scale while providing deterministic guarantees as the switch fabric scheduler is centralized and complex.

Reduction in scheduling complexity makes CIOQ routers practical. The possible deterministic performance guarantee makes this architecture interesting for core routers where a high throughput is crucial.

Many routers today allow adding line cards incrementally and independently. These line cards can even support different protocols and have different features. As these line cards may process packets differently they need to store cells locally on the input part [11]. The CIOQ router meets also this requirement.

CIOQ is today the most popular architecture for high-speed routers. ”Cisco Systems, currently the world’s largest manufacturer of Ethernet switches and IP routers, deploys the CIOQ architecture in Enterprise, Metro, and Internet core routers” [11], e. g., the Cisco CRS-1 (Carrier Routing System) [22, 23].

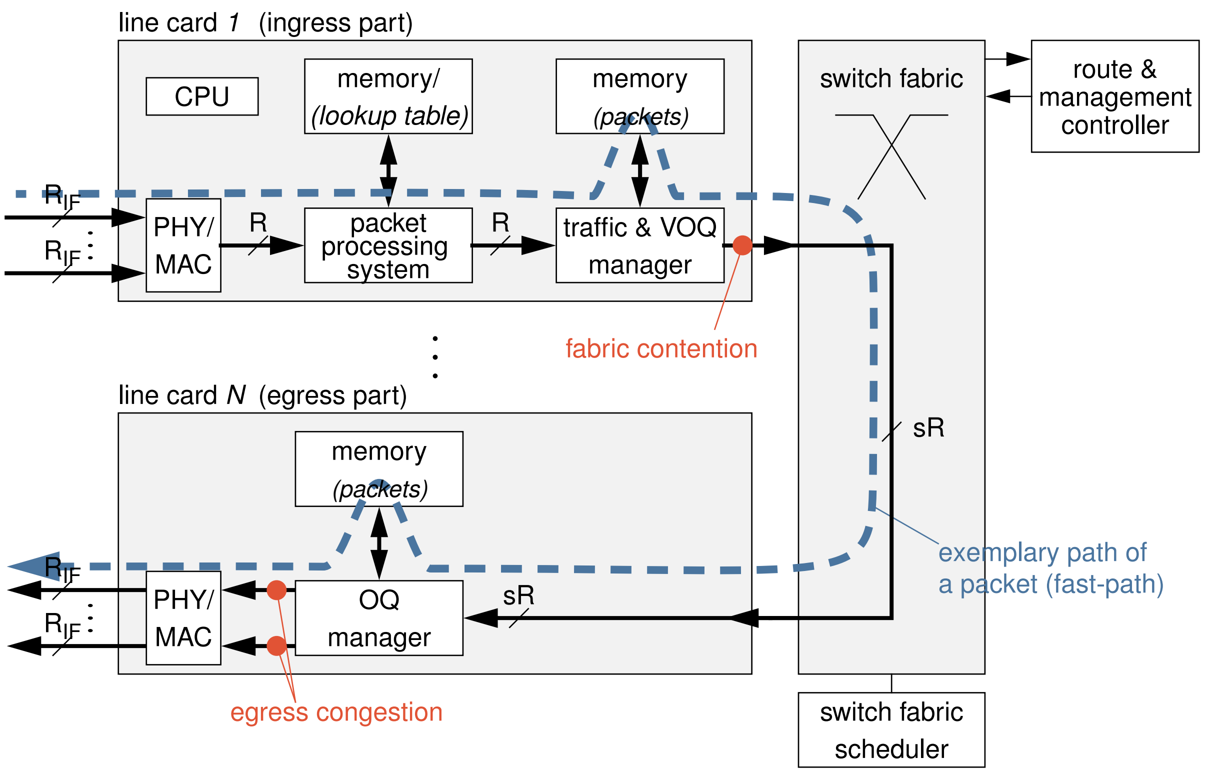


Figure 2.7: Basic Combined Input/Output-Queued IP Router architecture

2.1.4 Combined Input/Output-Queued Router

The most popular type of router used in the Internet today is the CIOQ router as it simplifies switch fabric scheduling (cf. the previous Section). Figure 2.7 shows the basic architecture of such a CIOQ IP router.

The tasks performed by an IP router can be classified into two categories: control plane tasks and data plane tasks [24].

Control plane tasks include system configuration, management and exchange of routing table information. These tasks are performed relatively infrequently compared to normal packet processing in the data plane, but are often complex. As throughput is not crucial here, general purpose CPUs (Central Processing Unit) perform processing, e. g., CPU on ingress line card, CPU on Route and Management Controller (RMC) card. The path of the packets through the control plane modules is also referred to as *slow-path*. The RMC handles packets that need extra attention, participates in routing protocols, reserves resources etc., while the CPU on the line card performs local forwarding table updates, exception handling, etc.

Data plane tasks are the typical tasks a router performs on nearly all incoming packets, e. g., classification, lookup the forwarding table, packet header modification and queuing. To support processing at line rate the corresponding modules are implemented with specialized hardware, e. g., by a Network Processor (NP), an Application Specific Integrated Circuit (ASIC), or a Field Programmable Gate Array (FPGA). The path of the packets through the data plane modules is also referred to as *fast-path*.

Since packet buffering is a data plane task, in the following the modules along the fast-path are described in detail (cf. Figure 2.7). Contents of the descriptions partly originate from [24].

PHY/MAC – The PHY chip receives the optical signal from the fiber and performs opto-electronic conversion, as well as clock and data recovery. The MAC chip detects the frame boundaries, checks the integrity of the frame and delivers it to the packet processing system.

Packet Processing System – This module is a processing system that is specialized on packet processing. For example, it usually contains processing units with a specialized instruction set architecture and highly parallel processing. The packet processing system parses and classifies incoming packets to isolate the individual flows. Packets destined for the slow-path are forwarded correspondingly. For fast-path packets it performs a route lookup to determine the outgoing interface, and modifies certain header fields. Depending on the lookup algorithm different off-chip[4] memory technologies are used to store the lookup table. During processing, the packets are stored in a small on-chip memory. Finally, it forwards the packets to the traffic and VOQ manager module. A commercial example for a packet processing system is the Silicon Packet Processor (SPP) [22] of Cisco, which operates on their 40 Gbps CRS-1 router line cards. Hauger presents in [25, 26] a novel architecture for high-speed packet processing systems. This provides high speed and flexibility to adapt to new requirements simultaneously.

Traffic Manager – This module ”is responsible for prioritizing and regulating the outgoing traffic” [15] according to the service level agreements with the individual subscribers. Therefore it monitors the traffic and may also take corrective actions by policing the traffic via dropping, delaying or marking packets. Chao *et al.* present and discuss in [24] corresponding algorithms.

VOQ Manager – The Virtual Output Queue manager implements a packet buffer for temporary storage of packets to resolve contention among the different inputs of a switch fabric, i. e., on several input line cards packets may be available which are destined to the same output line card. The VOQ manager organizes the packets in queues to avoid head of the line blocking. These queues are called VOQs. One or more flows may share a queue in the buffer. It typically maintains *number of output interfaces* × *number of CoS* queues [27], i. e., each queue corresponds to an output interface and a class of service. In real high-speed systems this leads to hundreds to thousands of VOQs [24, 27, 11]. A *memory manager* maintains the individual queues and organizes the required control structures. The memory manager may also keep some overall statistics that can be used by the traffic manager, e. g., memory occupancy of the individual queues, memory occupancy of a group of queues, etc. A packet memory stores the individual packets. To enable large capacities, this memory is typically implemented with off-chip memory modules.

Switch Fabric Scheduler – Assuming a router with a bufferless crossbar based switch fabric the router also contains a centralized switch fabric scheduler. This schedules the cell

[4]Off-chip memory is memory that is implemented on a different chip than the data processing task. E. g., the memory modules in a personal computer are off-chip memories. External memory is a synonym for off-chip memory. In contrast to this, on-chip memory is memory that is integrated into the same chip like the data processing or memory management task. Internal memory is a synonym for on-chip memory. Typically on-chip memory can be accessed much faster compared to off-chip memory.

transfers from input line cards to output line cards via the switch fabric. At the beginning of each cell time, the scheduler selects a configuration for the switch fabric and then transfers cells from switch fabric input to output. To determine the switch fabric configuration the scheduler implements a matching algorithm (e. g., iSLIP [21]) that may take account of queuing information (e. g., queue length), CoS, and statistics maintained by the line cards.

Switch Fabric – The switch fabric interconnects the individual line cards as discussed in Section 2.1.3.1. Switch fabrics often implement a speedup to account for fragmentation of the cells and the non-optimal matching algorithms implemented by the switch fabric scheduler. The speedup s is the number of times a module works faster then than the line rate, e. g., $s = 1$ means there is no speedup, $s = 2$ means the module works twice as fast as the line rate. Accordingly, cells depart from the input line card and arrive on the output line card with a rate of sR.

OQ Manager – The Output Queue manager implements a packet buffer for temporary storage of packets to resolve egress congestion and to match the speed gap between the switch fabric rate (sR) and the outgoing interface rate (R_{IF}). The number of output queues in high-speed systems is large because these often maintain one queue per flow, e. g., up to of ten thousands or even more [11]. Further, the OQ manager is similar to the VOQ manager except that a local scheduler serves the output ports based on their line rates. A huge number of potential algorithms are available in literature, e. g., weighted fair queuing [28] and its variants.

Summarizing, a router with CIOQ is a complex system with many modules. The packets traverse the modules along the fast-path from the input ports to the output ports in a fixed order, i. e., in a kind of pipelined fashion. All modules along the fast-path operate at line rate or even faster and therefore require specialized hardware. Along the fast-path there are two large packet buffers which are implemented by the queue managers, i. e., VOQ manager and OQ manager. In high-speed systems the packet buffer on the ingress part of the line card maintains a medium number of queues in the order of hundreds to thousands, while the packet buffer on the egress part of the line card maintains a larger number of queues in the order of ten thousands and more.

2.2 Packet Buffers

Internet routers and switches require high-speed packet buffers to hold packets during times of congestion. In the following, Section 2.2.1 defines the term *packet buffer* and gives an overview how it interacts with other components. Then Section 2.2.2 introduces the basic architecture and building blocks of a packet buffer. Section 2.2.3 explores the design choices and gives a practical example for queue management. Finally, Section 2.2.4 derives performance and capacity requirements for packet buffers.

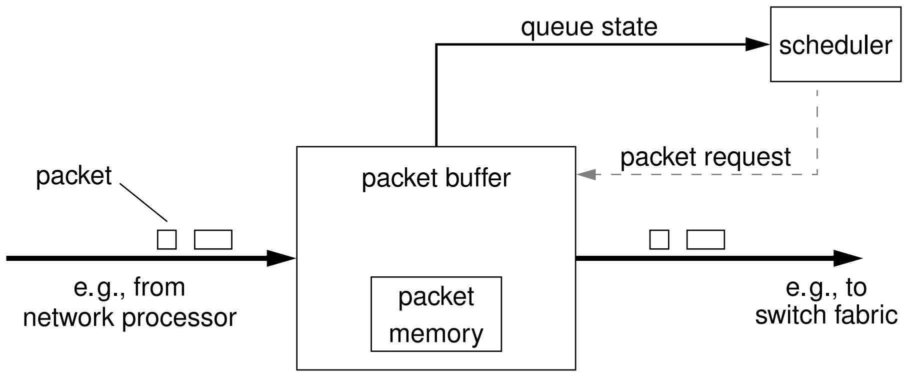


Figure 2.8: Packet buffer interaction with other components

2.2.1 Overview

In literature there is no clear definition of the term *packet buffer*. Authors use a wide range of expressions and wordings to describe packet buffering and the involved components. This Section gives a definition which is used throughout this thesis. Further, this Section introduces the interaction of the packet buffer with other components. Figure 2.8 shows the packet buffer and its connections to related components.

A **packet buffer** performs all tasks related to the buffering of packets: it abstracts from the underlying technology and from any queue and memory management.

High-speed packet buffers in routers always maintain several FIFO queues [11] in which they buffer packet data. To implement these queues it manages the packet memory. Thereby it may assign buffer memory to the individual queues statically or dynamically. Depending on the location of the packet buffer in the router architecture, these queues represent VOQs or OQs. A packet buffer may operate on packets or cells depending on its implementation.

From the packet buffer's point of view all data of one queue belong to one logical flow. For the remainder of this thesis the term *flow* is defined as follows: all packets that should be buffered in the same queue belong to one flow. For example, for an input buffer all packets with the CoS i destined to output line card j belong to one flow.

In the following the queues maintained by a packet buffer are named *flow queues*. The preceding component (e. g., an NP) delivers packets with meta-information. The meta-information contains at least the packet length and the flow the packet belongs to. Based on this the packet buffer allocates the right amount of memory and enqueues the packet into the corresponding flow queue. The packet buffer provides queue state and scheduling information to the scheduler, e. g., queue empty/non-empty and packet sizes of the head packets, respectively. Based on this information the scheduler requests packets from the packet buffer.

Summarizing, the tasks of a packet buffer are

- Buffering of packet data
- Queue management (enqueue incoming and dequeue requested packets or cells)

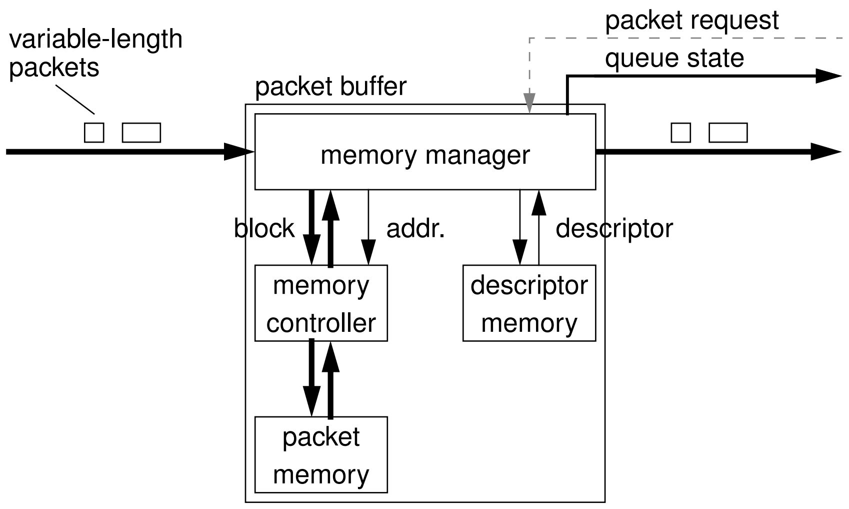


Figure 2.9: Basic architecture of a packet buffer

- Memory management (statically or dynamically allocate memory for queues)
- Provide queuing information to scheduler

The scheduler, also often named arbiter, requests packets or cells from the packet buffer. Its decision bases on the queue state information provided by the packet buffer (cf. Figure 2.8).

The following Sections focus on the building blocks and operation of a packet buffer according to the definition given above.

2.2.2 Basic Architecture

The basic architecture of a packet buffer is shown in Figure 2.9. In the following the individual building blocks are introduced. These are namely: packet memory, memory controller, memory manager, and descriptor memory.

Packet Memory – The packet memory stores the individual packets. In most cases the required capacity is too large for on-chip memory. Therefore usually off-chip memory is used in form of DRAM as this provides large capacities at low prices, e. g., DDR3 SDRAM [29].

Memory Controller – The memory controller abstracts from the potentially complex interface of the memory module to implement the packet memory. For example it assures that timing constraints of the memory device are met and it sends special required control sequences to initiate write or read process. Further it may optimize the throughput by exploiting memory characteristics, e. g., perform out-of-order reads and writes to minimize signaling overhead. The authors in [30, 31] show strategies how to increase DRAM bus utilization in packet buffers.

Memory Manager – The memory manager abstracts from actual memory organization and provides Q logical FIFO queues where packets are stored and retrieved from. There are many degrees of freedom maintaining these queues. The main design choices are discussed in Section 2.2.3.

Upon packet arrival, the memory manager allocates free memory in the packet memory. Then it triggers packet storage by providing memory address and packet data to the memory controller. In parallel it generates one or more descriptors which contain information about the packet, e. g., location (memory address) and packet length. It stores the descriptors in the descriptor memory and also links it to the tail of the corresponding queue. The memory manager provides the queue states to an external scheduler.

Upon a packet request from the scheduler, the memory manager accesses the descriptor memory and retrieves from the corresponding queue head the one or more descriptors belonging to the packet. Knowing packet length and location it triggers packet retrieval from packet memory.

Descriptor Memory – The descriptor memory stores the individual descriptors. Its size is usually proportional to the packet memory size[5] as each packet or portion of data requires a descriptor. The queue operations (enqueue, dequeue, etc.) performed by the memory manager often require accesses that depend on each other, e. g., result of read access A determines address of read access B. Consequently, the descriptor memory requires a short access latency to enable fast queue operations. Usually, it is implemented by off-chip SRAM as this provides very short and constant access latencies. As SRAM provides a very simple interface, the corresponding memory controller is omitted here.

Concluding, the memory manager is the central component in packet buffer as it performs the complete queue and memory management, i. e., it adds and removes packets from the individual queues as well as allocates and de-allocates required memory. The degrees of freedom it has maintaining these queues are discussed in the following Section.

2.2.3 Memory Organization

This Section introduces how the memory manager actually organizes the packet memory. Therefore it shows available design choices for memory allocation and block size. Further, it presents the application of linked lists as state-of-the-art control structure along with its degrees of freedom, optimizations to reduce the control overhead, and an example.

Memory Allocation

The memory manager assigns memory *statically* or *dynamically* to individual queues. In case of *static memory allocation* each queue is statically assigned a memory area. This greatly simplifies implementation as the memory area can be organized as a ring buffer. However, this disallows sharing the available memory capacity, e. g., if a queue already occupies its full memory range further packets destined for this queue are dropped, even if there is enough unallocated space in the packet memory. Due to its very inefficient memory capacity utilization this is not a practical solution for typical network nodes.

[5]*size* is used in the following interchangeably for *capacity*

In case of *dynamic memory allocation* memory is dynamically assigned to the individual queues. To prohibit queues from monopolizing the packet memory dynamic partition mechanisms [32, 33] can be used to regulate queue lengths.

Block Size

In case of dynamic memory allocation, the memory manager divides the available memory into blocks. These blocks can be *variable* or *fixed* in size. With *variable-size blocks* no internal fragmentation occurs as the blocks exactly match the size of the packets. However, this has two main drawbacks. Firstly, variable-size blocks may lead to heavy external fragmentation, i. e., new blocks may not fit into the free memory fragments. Secondly, searching for free memory to store a new block is complex and challenging at higher line rates.

With *constant-size blocks* internal fragmentation occurs, i. e., a packet usually does not fill a block entirely. To minimize fragmentation a small block size is used, e. g., 64 byte is a common choice. The problem of finding free blocks gets trivial when blocks are of constant size.

Alternatively, the memory can be partitioned statically or dynamically into areas where each area has a different block size [34], e. g., four areas with the block sizes 64 byte, 128 byte, 512 byte, and 2048 byte, respectively. Packets that are larger than a block are segmented into several blocks. For example, a 550 byte packet occupies one 512 byte and one 64 byte block.

Control Structure

Assuming constant-size blocks and dynamic memory allocation there are many possibilities to implement the control structures for queue management, e. g., linked lists of blocks, pointer arrays, and binary trees. Wuytack *et al.* give in [35] an overview about control structures. Linked lists are flexible, resource efficient and simple to realize in hardware. Therefore, this is the typical control structure used. The following paragraphs introduce in detail how linked lists are used for queue management.

The memory manager segments incoming variable-length packets into constant-size blocks and allocates the corresponding number of blocks in the packet memory. For each allocated block it generates a block descriptor that contains information about the packet length, a pointer to the next block and information if this is the last block of a packet. To be able to read blocks of a single queue with line rate, the read latency to access the individual descriptors has to be low. This is mandatory as each block descriptor contains the pointer (memory address) to the next descriptor. Placing the block descriptors into the packet memory is very economic as no separate descriptor memory is needed. This also minimizes the number of I/O pins. Nikologiannis *et al.* propose in [36] a corresponding packet buffer architecture. However, this is not feasible in high-speed systems with very high packet rates. The reason therefore is that the DRAM technology used to implement the packet memory has a relative long and variable read latency what does not meet the requirements for descriptor memory. Systems targeting a high throughput [37, 38, 39, 40, 41, 42] store descriptors in a dedicated descriptor memory, which is implemented as a physically separate memory with short read latency compared to the packet memory.

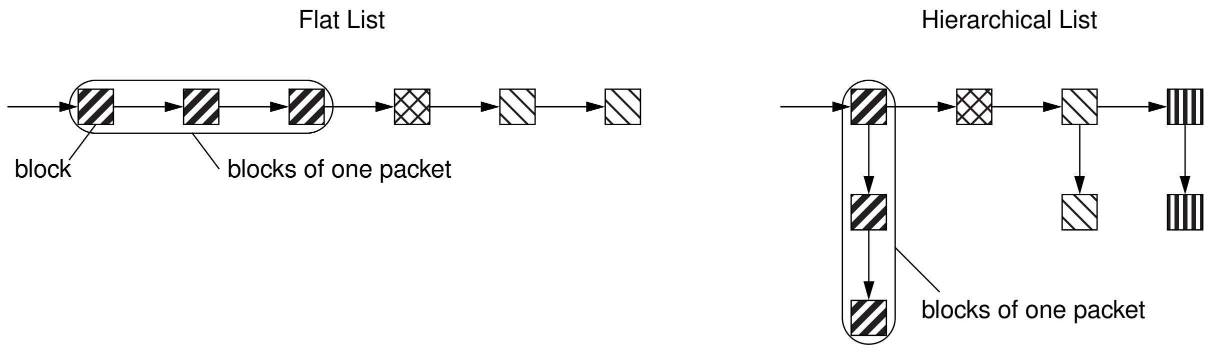


Figure 2.10: Flat and hierarchical linked lists like used in packet buffers

High-speed systems allow only strict FIFO access to the flow queues maintained to achieve a high performance. As in such systems all packet processing and manipulation takes place before the packet is enqueued this is no drawback.

Information about actual products is very rare, as vendors do not publish their implementations in such detail. One example with some information is the IXP2400 network processor from Intel [43] which provides hardware supported queuing. It uses relatively large block descriptors of 32 byte, which contain information about both, the block and the packet. With these descriptors the memory manager can create flat and hierarchical lists (cf. Figure 2.10). A flat list means a simple linked list of blocks. A hierarchical list means, each packet is represented by a linked list of blocks while the first blocks of individual packets are additionally linked together to a packet list.

To remember the head and tail of each queue the memory manager maintains a queue table which contains one queue descriptor per queue. A queue descriptor contains at least one pointer to the head block and one pointer to the tail block descriptor as well as a valid bit to mark the validity of an entry. As the queue table is accessed very frequently it is typically implemented with a dedicated memory. Due to its small size this memory can be usually implemented as on-chip memory.

To reduce the number of memory accesses often *preallocation* [36] is implemented: one free block is preallocated for every queue and linked as the last block in the queue. When a new block is to be enqueued, it is always written into the preallocated block. This corresponds to a write operation to the packet memory. Simultaneously, a new free block is linked to this block, which is thus preallocated. This corresponds to a write operation to the descriptor memory.

To keep track of free blocks the memory manager organizes them as an own queue which is often called the *free-list*. To reduce the access rate to the descriptor memory a free-list cache can be used which caches several pointers to free blocks. With a balanced packet arrival and departure to the packet buffer, a free-list cache greatly reduces access rate to the descriptor memory. To optimize the packet memory accesses, the reuse of free blocks can be adapted to the properties of the memory module used to implement the packet memory, e. g., Last-In First-Out (LIFO) reuse, FIFO reuse, one free-list per memory area or bank, etc.

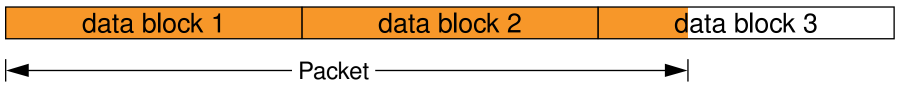


Figure 2.11: Packet segmentation to constant-size blocks

Reduction of Descriptor Memory Size and Bandwidth

As each data block requires a block descriptor a large packet memory leads to a large descriptor memory. Nevertheless, there are several options to keep the descriptor memory size small.

- **Reduction of block descriptor size:** If the packet length or any other information is not required by the system in advance, this can be stored in the packet memory along with the packet data. E. g., in case of the packet length an end-of-packet bit in the block descriptor allows to stop reading at the last block of a packet without knowing its explicit length.

- **Reduction of the number of block descriptors:** With increasing block size the number of block descriptors decreases. However, depending on the packet size distribution this may increase the internal fragmentation in the blocks dramatically.

- **Grouping of blocks to pages:** N adjacent blocks can be grouped to a page, while the packet memory is still accessed in granularity of blocks [44]. One page is explicitly allocated to one queue. Now only one page descriptor is maintained instead of N block descriptors. Inside a page the blocks are occupied sequentially and therefore require no explicit descriptor. This is a trade-off between packet memory fragmentation on the one hand and descriptor memory size and bandwidth on the other hand.

  Fragmentation occurs as the head and tail page of each flow are only partially occupied, i. e., the memory manager fills the head page sequentially and also empties the tail page sequentially. Non-occupied page parts cannot be utilized by other flows as a page is exclusively assigned to a single flow. For a moderate page size (e. g., 16 blocks) and number of flows the introduced fragmentation is negligible with respect to the extreme capacities provided by today's DRAMs.

  When the page size is large enough the size of the descriptor memory may drop below a value that enables implementation as an on-chip memory, saving I/O pins and costs. However, with increasing page size the page descriptor size is also growing as it contains now information from several packets, e. g., packet lengths, begin and end information of the individual packets.

  Further, using pages reduces the required bandwidth to the descriptor memory in average by factor N. The memory manager derives the memory address of the next block by a simple increment of the current address. So it accesses the descriptor memory only when it reaches the end of a page and wants to read or link the next page descriptor.

Example

Figure 2.12 shows the basic memory management with constant-size blocks and linked lists as control structure. Descriptor memory and packet memory are physically separate memories.

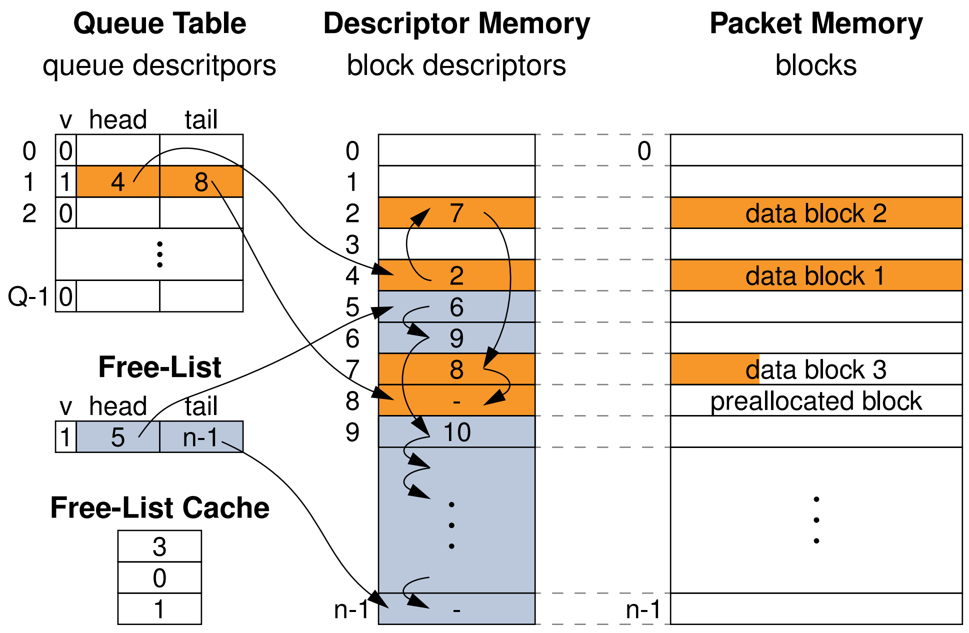


Figure 2.12: Basic memory management with constant-size blocks and linked lists as control structure

There is a one-to-one mapping between block descriptors and blocks, i. e., each block is associated to one fixed block descriptor. Therewith, the block descriptor does not need to contain the address of the block, as it is equal to the descriptor address, e. g., block descriptor 0 and data block 0 have both the address 0. Both, descriptor and packet memory contain n elements, i. e., the size of the packet memory is n blocks.

After powering on the systems the memory manager initializes the descriptor memory in a way that all blocks belong to the free-list. Every block, except the ones in the free-list cache, belong at any time to one of the Q queues or to the free-list.

The queue table contains Q queue descriptors. The head pointer points to the head block descriptor of the queue while the tail pointer points to an empty preallocated tail block descriptor.

When the memory manager has to read the block at the head of queue 1 in Figure 2.12 it performs the following tasks:

1. It accesses the queue table at address 1 to read the queue descriptor which contains the head pointer. Here the value of the head pointer is 4.
2. It accesses the packet memory at address 4 to read the corresponding block.
3. Simultaneously to 2., it accesses the descriptor memory at address 4 to read the block descriptor that contains the pointer to the new head. Here the value of the new head pointer is 2.
4. It accesses the queue table to update the head pointer in the queue descriptor to the new head. Here the value of the new head pointer is 2.

When the memory manager has to enqueue a block to queue 1 in Figure 2.12 it performs the following tasks:

1. It accesses the queue table at address 1 to read the queue descriptor which contains the tail pointer. To tail pointer points to the preallocated tail block. Here the value of the tail pointer is 8.
2. It accesses the packet memory at address 8 to write the corresponding block.
3. Simultaneously to 2., it fetches a free block from the free-list cache (e. g., block 3) and accesses the descriptor memory at address 8 to store the new block descriptor. This contains the pointer to the new preallocated tail block. Here the value of the pointer is 3.
4. It accesses the queue table to update the tail pointer in the queue descriptor to the new tail. Here the value of the new tail pointer is 3.

Summary

Summarizing, the memory organization offers several main design choices, e. g., blocks size and control structure. However, typically constant-size blocks with linked lists as control structure are used. The reasons therefor are that this is resource efficient, hardware implementation is relatively simple, and implementation offers a large number of design choices.

2.2.4 Requirements

A packet buffer has to fulfill requirements on the functional level and on the technological level. Further, the performance, i. e., here bandwidth, delivered by a packet buffer can be deterministic or statistical. This Section introduces the requirements and discusses the pros and cons of statistical and deterministic bandwidth guarantee.

Functional Level

The global requirement on the functional level is to **provide and manage a set of queues** to buffer user packets. Thereby it completely abstracts from the underlying memory technology and its organization. The previous Sections introduced in detail the architectural building blocks necessary therefore.

Technological Level

On the technological level a packet buffer has to satisfy requirements concerning three major properties: bandwidth, capacity, and access time. These requirements are mainly defined by system properties like the input line rate R to the packet buffer and the applied network technology, e. g., Ethernet [802.3] or Provider Backbone Bridging [802.1ah]. These requirements are quantified in the following.

Bandwidth – The packet buffer has to both, store and retrieve packets at line rate. Therefore, the minimal necessary memory bandwidth is 2R, where R is the aggregate line rate of all ports on a line card.

It is common for network nodes to segment packets into constant-size blocks to simplify memory management. 64 byte is a common choice as it is the first power of two able to hold a minimal Ethernet or IP packet. Nevertheless, in the worst case the packet buffer must sustain a stream of 65 byte packets, which all consume two 64 byte blocks, leaving the second nearly empty. This effect is called the 65-byte-problem [45].

Because of this effect memory bandwidth is often overdimensioned by a factor of two, i. e., to $4R$. A high memory bandwidth can be achieved by using many memory chips in parallel. However, with increasing number of I/O pins to connect the memory chips the system costs increase quickly.

Capacity – The capacity dimensioning of packet buffers is discussed controversially in literature. As dimensioning is not part of this thesis the author relies on common dimensioning rules.

It is common to use a rule-of-thumb, commonly attributed to [46], to dimension the capacity of the packet buffer. This suggests, for TCP to work well, the buffer should be dimensioned to the bandwidth delay product, i. e., $RTT \times R$ where RTT is the round trip time between active end hosts. Assuming $RTT = 200$ ms and a line rate of $R = 100$ Gbps the required buffer size is 2.5 Gbyte. In [12] the authors show that for core routers, buffer size can be reduced to $RTT \times R/\sqrt{N}$, where N is the number of major active TCP flows. With 10.000 TCP flows this would reduce the buffer size to around 1 %. However, this rule-of-thumb is widely in use and it is likely to be also in future as network operators do not want to put the Service Level Agreements (SLA) at a risk [47]. For example, a cutting-edge 100 Gbps line card from Alcatel-Lucent for their 7750 IP router platform uses 1.5 Gbyte of DRAM for buffering [48]. This is enough to buffer around 1.5 Gbyte / 100 Gbps $\approx$ 120 ms of traffic. The 100 Gbps line card is already available as prototype at time of writing and uses the in house developed FP2 chipset that also includes a chip for queue management.

Following the proposal of [12] with the assumptions from above the required buffer size is 25 Mbyte. With off-chip memory this leads to small benefit as still many parallel memory chips are needed to achieve the required bandwidth. However, when this amount of memory could be realized on-chip this would eliminate the large number of memory I/O pins and thereby greatly reduce overall system costs. Such a price shift would let network operators rethink if they really need routers with packet buffer capacities that are dimensioned according the rule-of-thumb.

The 65-byte-problem also affects capacity as it leads to fragmentation. This again requires overdimensioning. Similar to the bandwidth, memory capacity can also be increased by operating more memory chips in parallel.

Access Time – The access time ”refers to the minimum amount of time that needs to elapse between any two consecutive” [11] accesses to the packet buffer. For example, as it takes 5.12 ns to receive a 64 byte packet at 100 Gbps, a stream of 64 byte packets requires an access time of $T = 2.56$ ns since packets arrive to and depart from the packet buffer. This directly translates to the memory subsystem in the packet buffer, i. e., the access time of the packet buffer is only as short as the access time provided by the implemented memory subsystem.

Bandwidth Guarantee: Statistical or Deterministic?

Delivering *statistical* bandwidth guarantee means, the bandwidth is not guaranteed and depends on the traffic pattern. Packets can be occasionally lost, i. e., the packet buffer cannot store an incoming packet although there is free buffer capacity because the packet rate is temporary too high. Similarly, requested packets may be not delivered at all, too late, or even out-of-order.

Delivering *deterministic* bandwidth guarantee means, that the packet buffer accepts every incoming packet and delivers every requested packet independent of the traffic pattern, i. e., it never drops a packet. However, when the packet buffer is full, incoming packets cannot be accepted. The deterministic property implies that no packets are lost, all packets are delivered in-order and after a constant time.

A deterministic behavior has many advantages over a statistical:

- *100 % bandwidth guarantee under any traffic condition*
  This means, that no one can do better and packet drops occur only when the buffer is full.
- *Very simple packet buffer interface*
  This leads to a simpler subsequent component as this does not need to deal with not delivered packets, variable read latencies and packets delivered out-of-order.
- *Safety against adversarial attacks and "bake-off" tests*
  This means, that there are no loopholes an adversary (e. g., a virus or a hacker) could exploit. Examples for loopholes are specific traffic patterns or even just a specific packet size that leads to a significant drop in router performance. Tester companies [49, 50] that are specialized on "bake-off" tests analyze network devices for these loopholes [11].
- *Support for protocols which are designed for a network which never drops packets*
  Examples for such protocols are data center Ethernet [51] or fiber channel.

A packet buffer with deterministic bandwidth can constantly deliver highest performance what is crucial in high-end routers like a core router. Inevitably this requires more resources than providing just a statistical guarantee.

Summary

A packet buffer has to provide and manage a set of FIFO queues while fulfilling the technological requirements bandwidth, capacity, and access time. The latter are dictated by the supported line rate R and the used network technology. Bandwidth and access time are hard requirements that a packet buffer has to fulfill to sustain a worst-case packet stream. Capacity is a soft requirement which has no sharp bound and is only relevant when considering higher layer protocol performance like TCP.

The required bandwidth and capacity grow linearly with the line rate R and the required access time decreases linearly with the line rate R. This fact poses big challenges to high-speed packet buffer design as memory technology cannot keep up with this. Finally, the bandwidth guaranteed can be deterministic or statistical. Deterministic behavior has many advantages, which come at the price of a higher system cost.

2.3 Memory Technologies

The previous Section introduced the building blocks and requirements of a packet buffer. To implement the packet memory and the descriptor memory the suitable memory technology has to be selected for each.

For efficient implementation of the huge number of memory accesses performed by a packet buffer a Random Access Memory (RAM) is required. RAM allows stored data to be accessed in any order. i. e., at random. ”The word *random* thus refers to the fact that any piece of data can be returned in a constant time, regardless of its physical location and whether or not it is related to the previous piece of data. By contrast, storage devices such as tapes, magnetic discs and optical discs rely on the physical movement of the recording medium or a reading head. In these devices, the movement takes longer than data transfer, and the retrieval time varies based on the physical location of the next item.” [52]

Modern RAMs are typically volatile storage, i. e., they require power to maintain the stored data. At time of writing two practically relevant types of volatile RAM exist: static RAM (SRAM) and dynamic RAM (DRAM).

This Section is organized in four Subsections. The first Subsection introduces important memory related terms. Subsequent Subsections introduce SRAM and DRAM along with their properties and variants. The focus is thereby on the principal architecture, operation, and properties of these memory types. Jacob *et al.* give in [53] an in depth introduction to memory technologies. This Section partly relies on this. Finally, the fourth Subsection faces packet buffer requirements with SRAM and DRAM properties.

2.3.1 Terminology

This Section introduces important memory related terms.

Bandwidth – ”This refers to the total amount of data that can be transferred from a single memory per unit time.” [11] The typical unit of measurement is gigabit per second (Gbps). The maximum or peak bandwidth refers to the maximum achievable bandwidth.

Capacity – ”This refers to the total number of bits that” [11] a memory can store. The typical units of measurement are bit and byte, e. g., Mbit, Gbit, Mbyte, Gbyte.

Read Latency – This refers to the time between issuing a read request to the memory and receiving the corresponding data. The typical units of measurement are number of clock cycles of the memory bus and the explicit amount of time, e. g., in nanoseconds (ns).

Access Time – ”This refers to the minimum amount of time that needs to elapse between any two consecutive” [11] accesses (read or write) to the memory. Its absolute value depends on the memory type. Further, it can be either constant or vary between a minimal and maximal value based on the relation of the consecutive accesses. The access time limits the possible number of read/write operations to the memory.

Random Access Time – This refers to the maximum of the access time and is thus a constant value. This time is used for worst-case evaluations.

2.3.2 Static Random Access Memory (SRAM)

This Section introduces SRAM. Therefore, it briefly presents the technology used to implement SRAM as well as the basic architecture of an SRAM device. Then, it introduces the individual properties of an SRAM related to its operation. Finally, it gives an overview of currently available and announced high-speed SRAM types.

Technology

SRAM is a type of volatile semiconductor memory. An SRAM memory cell that stores one bit ”is implemented as two cross-coupled inverters accessed using two pass transistors. This cross-coupled connection creates a regenerative feedback that allows the cell to indefinitely store a single bit of data.” [53] The word *static* in its name indicates, that its cells do not need to be refreshed periodically and that the stored information is not changed by the read operation.

There are several alternatives to implement a memory cell. Currently, most conventional designs use the full-CMOS (Complementary Metal Oxide Semiconductor) six-transistor memory cell, also called 6T memory cell [53]. The high number of transistors required per bit limits the capacity of commercial SRAM devices, e. g., currently typical SRAM chips have capacities of up to only 72 Mbit [54].

SRAM has two main advantages. Firstly, its performance is superior over other memory structures because the state of a memory cell can be written and read directly. There is no need to wait for a capacitor to fill up or drain. E. g., the random access time of SRAM is more then an order of magnitude smaller than that of DRAM. Secondly, SRAM uses the same fabrication process as required for the logic circuits of Network Processors, Queue Managers, etc. This simplifies integration of SRAM onto the Network Processor or Queue Manager die. This so called on-chip memory can be fitted in depth and word width to the specific requirements. Each on-chip memory can be placed on a die next to the logic circuit accessing it. This minimizes signal propagation times and thus increases performance.

Architecture

In an SRAM the individual memory cells are organized as an $x \times w$ array, where x is the number of words and w the number of bits per word. Figure 2.13 shows the basic architecture of an SRAM device.

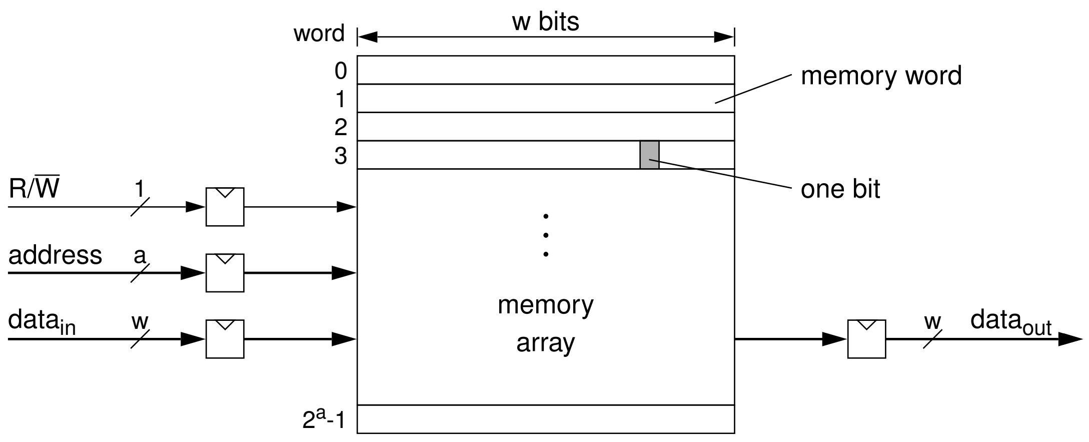


Figure 2.13: Basic architecture of an SRAM device

Its superior simplicity originates from the static behavior of the individual memory cells. These hold the stored information persistently as long as the memory is powered. Due to these properties SRAM provides a constant access time.

Operation

Now the individual properties of an SRAM related to its operation are introduced. For each property the degrees of freedom are introduced, that an SRAM manufacturer has for implementing a specific SRAM. The number of choices is large even if these are not all orthogonal to each other.

Access Method – Two access methods exist: asynchronous and synchronous. Asynchronous SRAM operates without a clock signal. Synchronous SRAM (sometimes abbreviated SSRAM) uses a clock signal, i. e., all timings are initiated by the clock edge and all signals are associated with the clock signal. As the logic circuit accessing the SRAM is also synchronous typically synchronous SRAM is used. Figure 2.13 depicts the architecture of a synchronous SRAM noticeable at the registers at the input and output interface.

Communication – The communication with a synchronous SRAM device can be *flow through* or *pipelined*. If *flow through* communication is used there is no overlapping between request processing and data transmission. E. g., a new read request can be issued earliest when the SRAM completely finish processing the last request.

If *pipelined* communication is used, the individual input and output signals to the SRAM are registered, i. e., registers are included into the communication path (cf. Figure 2.13). These registers increase the read and write latency but at the same time allow for higher clock rates.

Data Words per Clock Cycle – This refers to the number of data words received from or transmitted to the SRAM during one clock cycle. Current SRAM devices support either Single Data Rate (SDR) or Double Data Rate (DDR), i. e., one or two data words per clock cycle. With DDR at both, rising and falling clock edge, a new data word is exchanged.

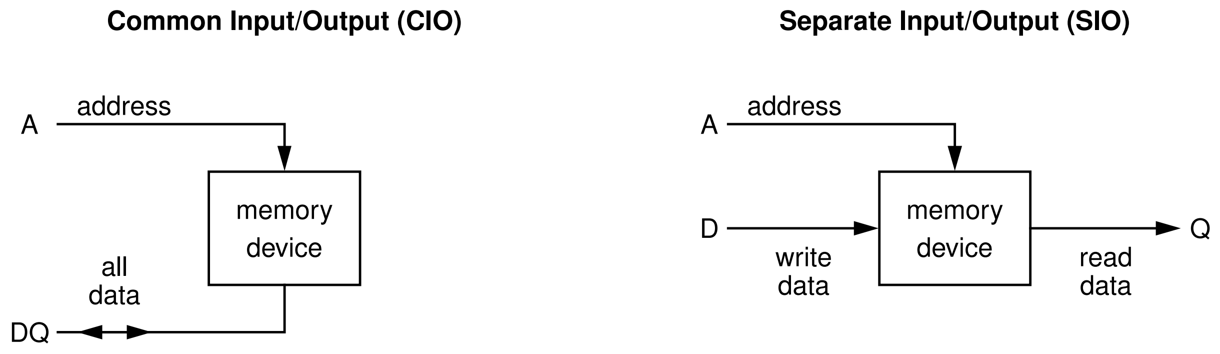


Figure 2.14: Memory data I/O types

Data I/O Type – The data input/output (I/O) type can be common or separate. Common I/O (CIO) means that the data bus is bi-directional, i. e., the data bus (DQ) is time multiplexed to transfer data words from and to SRAM (cf. Figure 2.14). Separate I/O (SIO) means, that two uni-directional data busses exist: one for read and one for write data (cf. Figure 2.14). With SIO there is no bus turnaround penalty. This maximizes the throughput for a read write ratio of 1. However, with only one type of requests (read or write) only one data bus is utilized and therefore throughput is bound to 50 %. In such a scenario an SRAM with CIO achieves 100 % throughput.

Burst Mode – Burst mode means that instead of a single data word a burst of data words is transferred with each request. The number of data words is called *burst length*. Burst mode intends to increase data bus utilization, in best case to 100 %. For example, an SRAM device typically accepts a read or write request only with the rising clock edge. To fully utilize the data bus (100 %) with DDR, a burst length of 2 is required. Another typical burst length is 4. This burst length is typically used in SRAM devices with SIO and a common address bus.

Memory Interface Type – The type of the memory interface can be *parallel* or *serial*. Nearly all commercially available SRAMs today have a parallel interface. Parallelism allows realizing very small read latencies. This can be mandatory depending on the application, e. g., for a descriptor memory.

Magnalynx [55] offers SRAMs with a *serial* interface. The main target is to reduce the number of I/O pins and thus the overall system costs. With fewer I/O pins per memory device also potentially more memory devices can be connected to a processing chip. However, serialization introduces additional latencies. If this is acceptable depends on the application.

Commands – An SRAM requires only two commands for operation: read and write. This is typically encoded with the one bit signal $R/\overline{W}$ (cf. Figure 2.13). $R/\overline{W}$ = '1' encodes a read operation, i. e., the memory word at the addressed memory location is returned after a constant read latency. The read latency is caused by the registers inserted into the communication path. $R/\overline{W}$ = '0' encodes a write operation, i. e., the data word on the data bus will be written to the addressed memory location.

Summarizing, independent of the degrees of freedom, the operation of an SRAM is very simple. This results from the two main properties that only two commands (read, write) are required for operation and that the read latency is constant.

Commercial High-Speed SRAM Types

At the time of writing, the only practically relevant high-speed SRAM types are provided by the Quad Data Rate (QDR) consortium [56], which consists of the memory manufacturers Renesas, Cypress Semiconductor, Hitachi, IDT, Micron Technology Inc., NEC and Samsung. The QDR consortium jointly defined and developed the following two off-chip SRAM types: Double Data Rate SRAM (DDR SRAM) and Quad Data Rate SRAM (QDR SRAM).

A competing consortium called SigmaRAM consisting of GSI Technology, Mitsubishi Electric, Sony Electronics, Toshiba and others seems to be discontinued since several years. The SigmaRAM consortium offered SRAM devices functionally equivalent to those available from the QDR consortium's members called SigmaDDR and SigmaQuad. GSI Technology [57] still offers these SRAM devices and also develops new versions according to their website.

Both, DDR SRAM and QDR SRAM are *synchronous*, *pipelined* and operate with *double data rate* and a *burst length* of 2 or 4. Regarding the I/O type both, CIO and SIO are available. DDR and QDR SRAM are available in the versions I, II and II+ while version III is announced for the near future on the manufacturers websites. Towards higher version numbers mainly operating frequency and memory capacity increase.

The two main differences between DDR and QDR SRAM are:

- QDR[6] SRAM is only available with SIO while DDR SRAM is available with CIO and SIO.
- QDR SRAM allows concurrent operations while DDR SRAM does not, i. e., a QDR SRAM with burst length 2 accepts one read and one write operation per clock cycle and so utilizes both of its data busses to 100 %.

Figure 2.15 shows for several SRAM types the data bus utilization as a function of the read and write operation ratio. The first letter in the legend indicates the type of SRAM (**D**DR or **Q**DR). B2 and B4 refer to burst length of 2 and 4, respectively. DDR SRAM with CIO achieves 100 % throughput when only read or only write operations are performed. Towards a read/write ratio of 1 the throughput decreases, while it reaches its minimum at 1. The reason for this is that there is a bus turn around penalty for the bidirectional data bus. The throughput of QDR SRAM behaves inversely to that of DDR SRAM, i. e., it reaches its maximum at a read/write ration of 1 as its two data busses are fully utilized simultaneously only at this point.

DDR and QDR SRAMs are targeted for different applications. DDR SRAM targets applications with unbalanced read write access, e. g., lookup tables that are updated rarely. QDR SRAM

[6]The name QDR SRAM origins from the fact that these devices are 4 times faster than devices with CIO and single data rate.

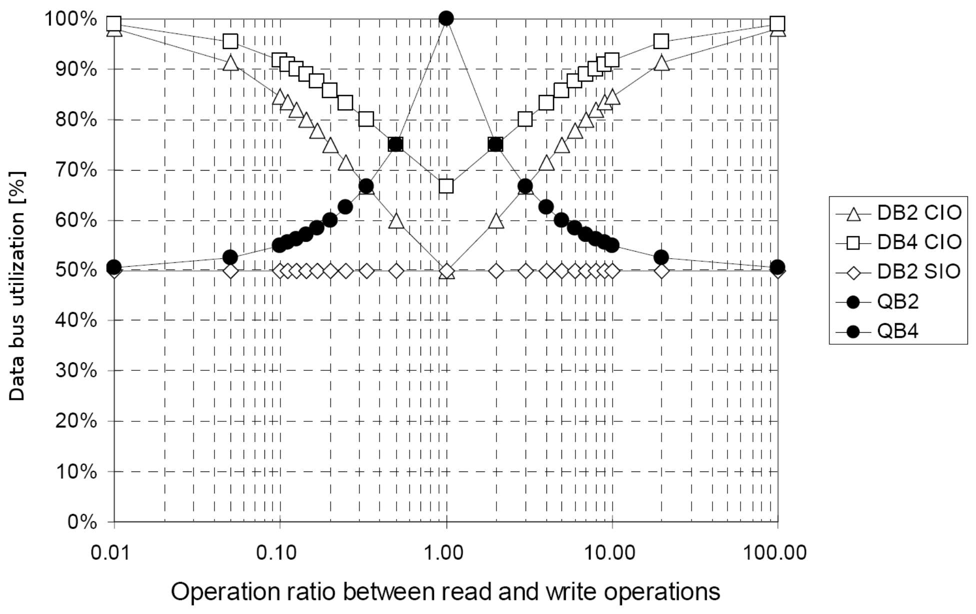


Figure 2.15: SRAM data bus utilization as function of the read and write operation ratio (Source: [58])

targets applications with balanced read write operation ratio, e. g., packet memory or descriptor memory.

Table 2.1 gives an overview of currently available and announced high-speed SRAM types and their properties.

SRAM Type	max. capacity / chip [Mbit]	max. clock freq. [MHz]	data pins / chip / bus [#]	random access time [ns]	burst length [words]	peak data rate / pin [Gbps]
DDRII (CIO) [54]	72	300	x9, x18, x36	3.33	2, 4	0.6
QDRII (SIO) [54]	72	300	x9, x18, x36	1.67	2, 4	0.6
DDRIII (CIO) [59]*	288	500	x9, x18, x36	2.00	2, 4	1.0
QDRIII (SIO) [59]*	288	500	x9, x18, x36	1.00	2, 4	1.0

Table 2.1: High-speed SRAM types and their properties; SRAM types marked with a * are not yet in production, but production is announced for the next years.

2.3.3 Dynamic Random Access Memory (DRAM)

This Section introduces DRAM. Therefore, it briefly presents the technology used to implement DRAM as well as the basic architecture of a DRAM device. Then it introduces the individual properties of a DRAM related to its operation. Finally, it gives an overview of currently available and announced high-speed DRAM types.

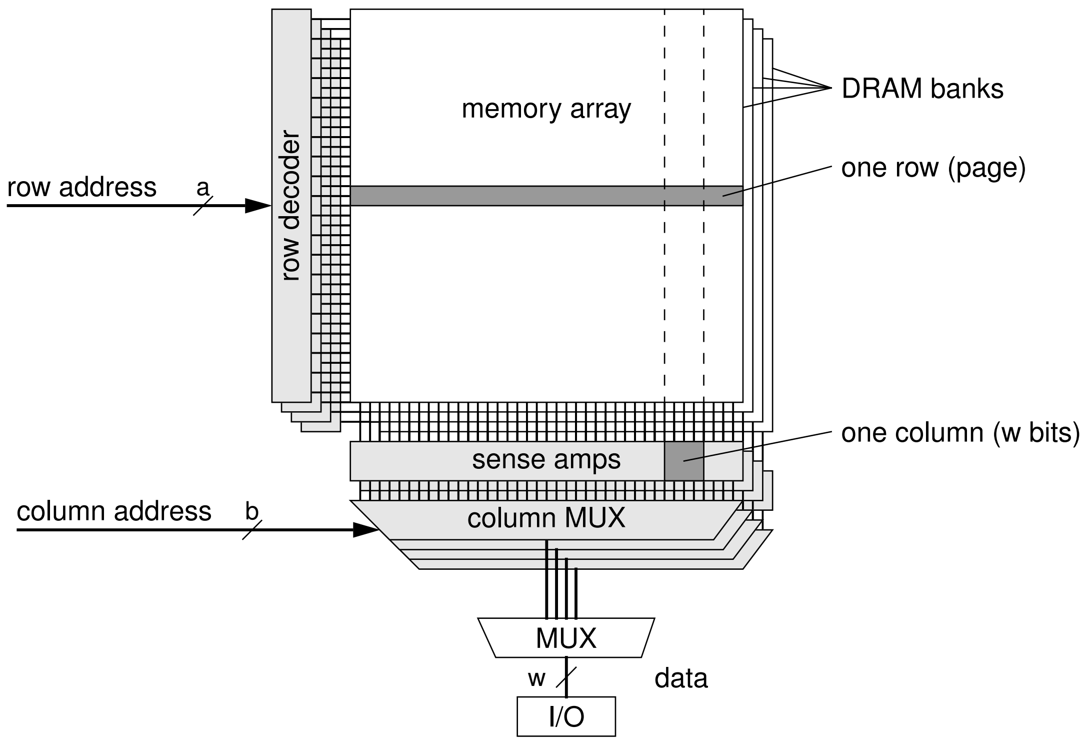


Figure 2.16: DRAM device architecture. A DRAM device consists of several independent banks. Each bank consists of the memory array and associated circuitry.

Technology

DRAM is a type of volatile semiconductor memory. A DRAM memory cell that stores one bit is implemented as a single transistor-capacitor pair. ”The circuit is *dynamic* because the capacitors storing electrons are not perfect devices” [53] and need periodical refresh. More detailed, the leakage of the capacitors necessitates this. To retain information stored in the DRAM, each of its capacitors must be periodically refreshed, i. e., read and rewritten [53]. Analogously, each read access to the DRAM discharges the corresponding capacitors. So to retain information each read has to trigger a rewrite operation.

In contrast to SRAMs, the dynamic property of DRAMs highly increases the complexity of their operation. Further, the time required for refreshes and rewrites after a read operation increase the access time significantly. However, the small footprint of a DRAM memory cell allows to build chips with huge capacities, e. g., DRAM chips with 4 Gbit are state of the art [54].

Summarizing, the large possible capacity in contrast to SRAM comes at the price of a complex device operation and increased access times. As it is shown in the following, manufacturers try to alleviate or even completely hide the long access times by providing clever DRAM architectures.

Architecture and Properties

The core of a DRAM is the memory array (cf. Figure 2.16). It is a rectangular grid of storage cells each holding a bit of data. This array is organized in *rows* and *columns*. An $8192 \times 512 \times 16$ memory array consists of 8192 rows, 512 columns while each column has a width of 16 bit. Such a memory array along with it associated control logic (sense amplifier and address decoders) is called a *bank*.

The access to a column requires two steps. Firstly, the row is addressed and its memory cells are sensed via the sense amplifiers. This is a special circuit used to detect the values stored on the capacitors. This step is called *opening a row*. The cell capacitors of the complete row are discharged by sensing. Secondly, the column of the sensed row is addressed and the data is transferred to the memory I/O. To access a column in another row of the same bank the data of the current row has to be rewritten and the internal circuits have to be prepared for the next sensing. This is called *closing a row*.

To hide the latencies introduced by opening and closing rows, vendors integrate several banks on a single DRAM device (cf. Figure 2.16). Since each bank has its own circuitry, the banks can operate independently of each other. This means, that while in one bank a row is open and the corresponding data is read, a row in another bank can be opened simultaneously in preparation for the next access. This *interleaving* of accesses allows in the ideal case to hide all latencies. For example, if a DRAM bank can deliver a new chunk of data every 40 ns, one can access 4 banks in a round robin fashion to deliver a new chunk of data every 10 ns. This quadruples the data rate achievable by any one bank. However, this requires that the accesses are distributed to the individual banks accordingly well.

The access time T of a DRAM has been defined Section 2.3.1 (page 27). For DRAMs, T depends on the sequence of memory accesses. Access sequences can be categorized into four groups [11]:

1. **Columns in the same row and bank:** This access sequence requires the smallest access time because the corresponding row is already opened and the columns can be accessed immediately. This is fast and delivers the highest bandwidth. However, it is not a typical access sequence in a router, as the requested data blocks usually reside in different rows and banks.

2. **Columns in different rows but in the same bank:** This access sequence leads to the longest possible access time because the current row has to be closed and the new row has to be opened prior reading the requested column. This worst-case access time for random accesses is named row cycle time (T_{RC}) in memory parlance. This is called a *bank conflict* or a *bank collision*.

3. **Columns in different rows and different banks:** This access sequence benefits best from interleaving banks, i. e., while reading from an open row in bank A, in bank B a row can be opened in preparation for the next access. With a sufficient number of banks this ideally leads to the same memory bandwidth as it has been achieved with the first access sequence category. However, to limit the current draw of a single DRAM the row to row activation delay (T_{RRD}, see Figure 2.18) sets a lower bound for the access time [53]. For

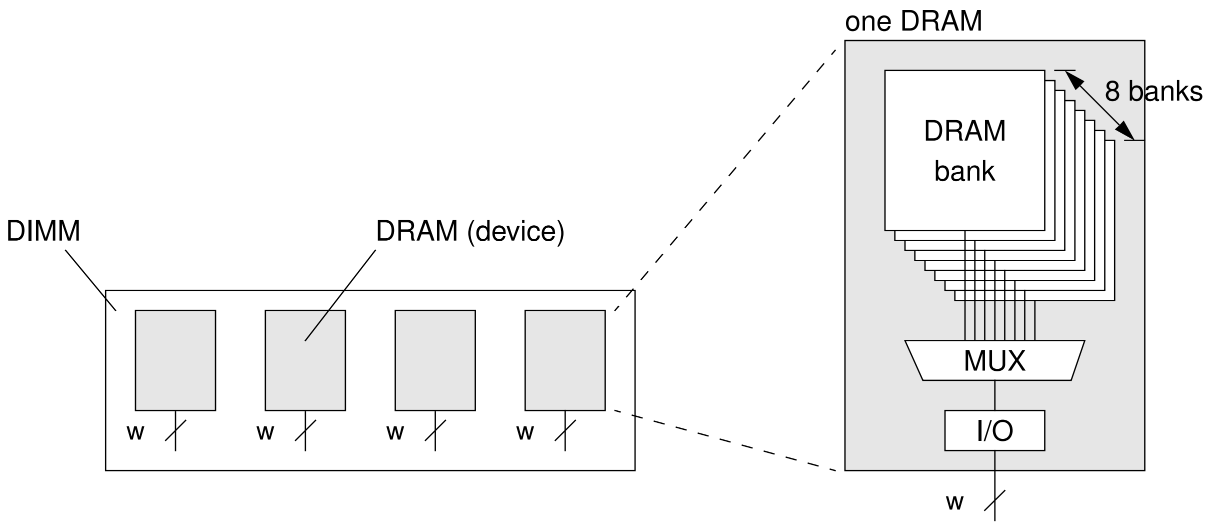


Figure 2.17: DRAM - levels of organization; Example shows a DIMM with 4 DRAMs, where each DRAM consists of 8 banks

the same reason, the four bank activation window (T_{FAW}) additionally raises the lower bound of the access time by allowing only four rows to be opened in this time window (T_{FAW}).

4. **Columns in adjacent banks:** ”Some DRAMs such as RDRAM [60] incur a penalty when two consecutive” [11] accesses ”are made to adjacent banks. This is called an *adjacent bank conflict*” [11] and occurs ”if adjacent banks share circuitry, such as” [11] sense amplifiers. ”Most modern DRAMs do not share circuitry between adjacent banks, and hence do not exhibit adjacent bank conflicts.” [11]

Manufacturers usually offer DRAMs for sale as individual memory chips, so called DRAMs or DRAM devices. Commodity DRAM, which is especially targeted in this thesis, is also offered as Dual Inline Memory Module (DIMM). A DIMM is a small Printed Circuit Board (PCB) with several DRAMs mounted inline to achieve a reasonable capacity and data bus width. Figure 2.17 shows such a DIMM[7].

Since a system can have multiple DIMMs, the word *rank* was introduced to distinguish between DIMM-level independent and bank-level independent operations. A ”rank is a set of DRAMs that operate in unison” [53]. A DIMM can contain one or two ranks.

Summarizing, DRAMs are complex and have relatively long access times. To enable the user to hide the access time by interleaving, modern DRAMs use various levels of organization. ”A system is composed of potentially many independent DIMMs. Each DIMM may contain one or more independent ranks. Each rank is a set of DRAM devices that operate in unison, an internally each of these DRAM devices implements one or more independent banks.” [53].

[7]The word *dual* in the abbreviation DIMM origins from the fact, that DIMMs have separate electrical contacts on each side of the module. Hereby wider memory bus widths can be implemented on a PCB. In contrast, Single Inline Memory Modules (SIMM) have on both sides redundant electrical contacts.

Operation

Now the individual properties of a DRAM related to its operation are introduced. For each property the corresponding degrees of freedom are shown.

Access Method – Older generations of DRAMs were asynchronous devices, i. e., used no clock signal for operation. Due to the many timing requirements of a DRAM their operation was very complex. This drawback was removed by introducing a synchronous DRAM interface, i. e., all interface signals are associated with the clock signal. Synchronous DRAM is abbreviated SDRAM. Throughout this thesis, the explicit access method of DRAM is not essential and therefore the name DRAM is used.

Data Words per Clock Cycle – This refers to the number of data words received from or transmitted to the DRAM during one clock cycle. At the time of writing most DRAM devices support double data rate, i. e., at both, rising and falling clock edge a new data word is transferred. However, Rambus [60] announced XDR2 devices with 16$\times$ data rate transferring 16 data words per clock cycle.

Data I/O Type – The data input/output (IO) type can be common (CIO) or separate (SIO), equivalently to SRAMs. Also equivalently to SRAMs, with CIO there is usually a penalty for changing the operation type from read to write and vice versa. Which I/O type is more suitable depends on the access behavior of the corresponding application.

Burst Mode – Burst mode means that instead of a single data word, a burst of data words is transferred with each request. The number of data words is called *burst length*. The burst length is often a memory parameter configured during power up of the DRAM. In general, with larger burst length data bus utilization increases, because the data bus stays less idle during the time a row in another bank is prepared for access.

To allow the address bus and DRAM core operating at a lower frequency than the data bus, modern DRAMs use a so called prefetch architecture. This means a single address request results in a burst of words on the data I/O pins. For example, DDR3 SDRAM has an $8n$ prefetch architecture [29], where n refers to the data I/O width of the DRAM chip, 8 to the burst length, and $8 \cdot n$ is the size of the burst sequence in bit. DDR3 SDRAM supports only a burst length of 8, with what it can achieve full data bus utilization.

There is a trend to larger burst lengths, which is expected to continue because of the increasing discrepancy between the increasing interface clock rates and the relatively static access time. E. g., DRAM bandwidth increases roughly by 30 % per year [61] which is achieved by higher clock rates at the interface while DRAM access time drops only 7 % per year [12].

Memory Interface Type – The type of the memory interface can be *parallel* or *serial*. Today, most commercially available DRAMs have a parallel interface. Parallelism allows realizing high throughputs at very small read latencies.

Devices with serial interface usually use several serial lanes in parallel. Serializing greatly reduces the number of required pins and thereby also cost. Examples are Fully Buffered DIMMs (FBDIMM, [62]) which are optimized for the use in high-end servers and Serial Port Memory Technology (SPMT, [63]) which is an industry standard memory interface

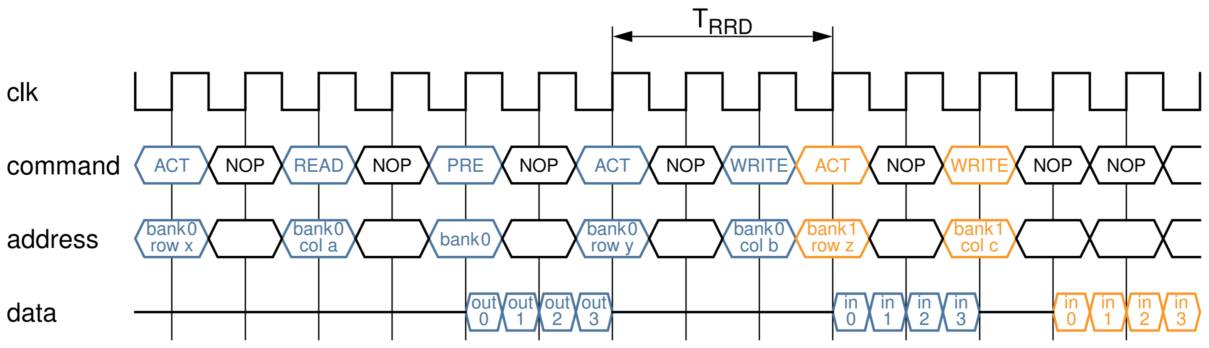


Figure 2.18: DRAM access to the same and to different banks; The access to the same bank but different rows is highlighted blue. The access to a different bank is highlighted orange.

designed for mobile and consumer electronics devices. However, serialization introduces additional latencies. If this is acceptable depends on the application.

Commands – DRAM requires a large number of commands for operation, e. g., a DDR3 SDRAM [29] supports 25 different commands. These can be classified into three categories: commands for configuration of the DRAM device, commands to enter/leave different power modes, and commands to access or preserve the data stored in the DRAM device.

In the following, a brief overview of the most important commands required to access the data is given. The DRAM-controller sends the *read* command to read a burst of words from an open row in a DRAM bank. The corresponding is also true for the *write* command. The DRAM-controller sends the *activate* command to open a row in a bank. To close a row it sends the *precharge* command. Figure 2.18 shows exemplary the required sequence of commands to access columns in two different rows of one bank. The third access, which is to a different bank, illustrates how the DRAM-controller can interleave accesses. It issues the corresponding activation (ACT) command earlier as banks are independent. An even earlier submission of this ACT command is not allowed because there is a time constraint between two activations (T_{RRD}).

To preserve the data in the DRAM, a periodical refresh is necessary. The corresponding *refresh* command first opens and then closes one or more rows in each bank. A DRAM-internal counter provides the row address. For example, in a DDR3 SDRAM [29] with 4 Gbit capacity each row has to be refreshed every 64 ms [29]. According to the standard [29] this requires 8192 refresh commands, i. e., on average one each 7.8 μs. Each refresh command refreshes several rows in each bank and takes 300 ns. From this follows that $8192 \cdot 300$ ns are required for refresh every 64 ms. This means that this 4 Gbit DRAM chip spends up to 3.84 % of its operating time with refreshing, what reduces the achievable bandwidth. Alternatively, the refresh command can be omitted when the DRAM-controller assures to open every row containing user data once every 64 ms. Depending on the application, this may require less or even no overhead for refreshing.

Summarizing, the operation of a DRAM is a complex task. Read/Write performance depends on the access pattern as well as on the optimizations carried out by the DRAM-controller, e. g., out-of-order execution and grouping of reads and writes to minimize penalties. Consequently, the access time also depends on the access pattern, but has an upper bound.

High-Speed DRAM Types

At the time of writing several types of high-speed DRAM are commercially available. However, due to the small networking market only one is explicitly specialized for networking applications. The following gives an overview about the most relevant high-speed DRAM types and their properties regarding the application in a packet buffer. It considers both, currently available as well as announced DRAM types.

The most important DRAM types are standardized by the Joint Electron Device Engineering Council (JEDEC) [64]. Standardization brings crucial benefits for router manufacturers: interoperability, competition and several suppliers. Here, the most important DRAM types standardized by JEDEC are the different generations of DDR SDRAM (Double Data Rate Synchronous Dynamic Random Access Memory) and GDDR SGRAM (Graphics Double Data Rate Synchronous Graphics Random Access Memory). DDR SDRAM is optimized for the use as main memory in PCs and is today present in most of them. Currently, generation 3 (DDR3 SDRAM) is in mass production while ratification of the DDR4 SDRAM standard is planned for 2011. GDDR SGRAM is also a dynamic RAM, but is optimized for graphical applications by supporting special operations[8]. Today it is present in most graphic cards. Generation 5 (GDDR5 SGRAM) is currently in mass production. Both use parallel and relatively wide busses to connect to the DRAM controller.

The largest competitor for the JEDEC DRAM types is the company Rambus [60] which defines its own memory standards. XDR DRAM is the newest generation high-speed DRAM being produced. The successor XDR2 is standardized and production is announced for the next years. Rambus targets with its XDR memory mainly high-performance gaming, graphics and multi-core compute applications. In contrast to JEDEC DRAM types RAMBUS uses a very narrow bus to connect the DRAM devices, running a computer-network like protocol on it [53]. Rambus holds the patents for its memory standards but is no manufacturer itself. In order to use Rambus memory, customers have to license the intellectual property for the whole memory architecture, i. e., memory controller, memory bus, memory devices.

Micron [65] offers a Reduced Latency DRAM (RLDRAM) that is optimized for networking applications, e. g., packet buffers, where short access times are required. Qimonda, the second manufacturer of RLDRAM, went bankrupt during the economical crisis in 2009. Micron produces currently the second generation of RLDRAM (RLDRAM II) and announced to start production of the third generation (RLDRAM III) in 2011. Compared to the aforementioned memories, RLDRAM has three big advantages: SIO, reduced access time and simplified interface (only few commands necessary to operate the memory device).

Table 2.2 gives an overview of high-speed DRAM types and their properties.

[8]GDDR SGRAM includes enhanced graphics features for use with display adapters. Its Block Write and Mask Write functions allow the frame buffer to be cleared faster and selected pixels to be modified faster.

DRAM Type	max. capacity / chip [Mbit]	max. capacity / pin [Mbit]	max. bus clock freq. [MHz]	data bus I/O type	data pins / chip / bus [#]	random access time [ns]	burst length [words]	data words /clk [words]	banks [#]	peak data rate / pin [Gbps]	relative price /bit [66]
DDR3 SDRAM [29]$^+$	8192	2048	1066	CIO	$\times$4, $\times$8	46.1	4, 8	2	8	2.13	1
GDDR5 SGRAM [67]	1024	64	2500	CIO	$\times$16, $\times$32	40.0	8	2	16	5.00	-
RLDRAM II [68]	576	64	533	SIO, CIO	$\times$9, $\times$18	15.0	2, 4, 8	2	8	1.06	20
RLDRAM III [68]*	1024	64	1066	CIO	$\times$18, $\times$36	10.0	2, 4, 8	2	16	2.13	-
XDR [69]	1024	512	800	CIO	$\times$2, .., $\times$16	-	16	8	8	6.40	-
XDR2 [70]*	-	-	800	CIO	$\times$2, .., $\times$32	-	-	16	-	12.80	-

Table 2.2: High-speed DRAM types and their properties; DRAM types marked with a * are not yet in production, but production is announced for the next years; $^+$ these are the maximal values standardized for DDR3 SDRAM. However, currently (spring 2010) the fastest commercially available DDR3 SDRAM features maximal 800 MHz and maximal 4096 Mbit; $\times N$ is the typical notation for the I/O data width of a DRAM chip

2.3.4 Discussion

A packet buffer usually contains a packet memory and a descriptor memory (cf. Section 2.2.2). Utilizing the techniques from Section 2.2.3 the *descriptor memory* is usually several orders of magnitude smaller than the packet memory. Consequently, the high-speed SRAMs presented in Table 2.1 suffice the requirements, as their main drawback is the small capacity.

In the following the requirements for the *packet memory* are summarized and compared to the available memory devices. The technological requirements of the packet buffer (cf. Section 2.2.4) directly translate to the requirements of the underlying packet memory. Consequently, the packet memory has to have a high bandwidth, a high capacity and a short access time. As absolute requirements we use the values required for a 100 Gbps Ethernet based system, as currently manufacturers announce the very first 100 Gbps routers to be available soon [48]. These requirements are a minimal bandwidth of 200 Gbps, a capacity of 2.5 Gbyte and an access time of 2.56 ns.

When looking for suitable memory devices we have to consider the following two facts. Firstly, we can increase bandwidth and capacity by simply operating more devices in parallel. Secondly, we cannot decrease the access time that simple when deterministic bandwidth guarantee is requested. Architectural approaches related to the access time will be discussed in Section 2.4.1.

According to Table 2.1 nearly all SRAM devices have a sufficiently small access time but a far too small capacity. Using parallel SRAM devices to achieve the required capacity is not practical. The high number of required devices would lead to extreme pin counts and space requirements. E. g., for 2.5 Gbyte a total of 320 QDRII SRAM devices are necessary.

Now, the DRAM devices in Table 2.2 are considered. All devices, suffice the **capacity** requirement of a 100 Gbps system by using a few up to some dozen devices in parallel. For example, DDR3 SDRAM has currently the highest capacity/pin ratio with 1024 Mbit/pin requiring just 5 devices in parallel with a total of 20 data pins for 2.5 Gbyte. With most other DRAM types around 300 data pins are required. This is a lot and therefore costly, but still practical.

To achieve the 200 Gbps **bandwidth** in a 100 Gbps system roughly 200 pins are necessary with any of the DRAMs. RLDRAM II has the lowest data rate/pin, but as it utilizes the memory bus more efficiently, it requires less overdimensioning.

The required access time for a 100 Gbps system is 2.56 ns, and thus an order of magnitude below the random access time of any DRAM in Table 2.2. Regarding the **access time**, for a 100 Gbps system all DRAMs in Table 2.2 fail by roughly an order of magnitude. Consequently, plain DRAMs are not suitable for high-speed packet buffers. Looking one step into the past one observes that this problem is relatively new, as it got first relevant with the introduction of 10 Gbps systems. E. g., in a 10 Gbps Ethernet based system the required access time is 25.6 ns. Comparison with Table 2.2 shows, that depending on the utilized DRAM the access time is either met or failed by a factor of just 2.

To evaluate the future trend we compare the DRAM properties to the requirements given by Ethernet[9]. The 100 Gbps Ethernet standard was ratified in June 2010. Hauger *et al.* estimate in [2] ratification of 1 Tbps Ethernet for 2015. This translates to a yearly increase of the required bandwidth and capacity by 58 % and a yearly decrease of the required access time by 37 %. Comparing this to the approximate increase in DRAM bandwidth of 30 % per year [61], the gap to the deployed line rates will increase continuously, requiring more parallel memories. Furthermore, the access time of DRAMs decrease by only 7 % per year [12] what drastically increases the gap. However, there are architectural concepts to cope with this access time discrepancy. These are addressed in the next Section.

As the only specialized DRAM memory for networking applications, RLDRAM has several advantages over others:

- *SIO*: the separate buses for input and output match to the balanced read write ratio in high-speed packet buffers, i. e., packets are written once to and read once from memory
- *Significantly reduced access time*: the memory can be accessed more frequently
- *Simplified interface*: this greatly simplifies implementation of the corresponding memory controller because timings are simplified and only a few commands are necessary to operate the memory device

Due to these advantages, Alcatel Lucent utilizes in its cutting-edge 100 Gbps router [48] RLDRAM II.

Beside these clear technical aspects, there are also others that are relevant for router manufacturers. These are given in the following.

- Independent of its cutting-edge performance GDDR SGRAM is disliked by router manufacturers. One reason for this is that product cycles of GDDR SDRAM are short, but router manufacturers have to provide long term support for their products.
- Rambus XDR DRAM is as well disliked because of the need of licensing intellectual property for the memory architecture. This is both, costly and requires tight cooperation with Rambus during development of the router.
- RLDRAM is usually the favorite choice due to its simplicity and short access times. However, it has two big drawbacks. Firstly, it is roughly 20 times more expensive per bit [66] than commodity DRAM (DDR3 SDRAM). This is an essential economic drawback since packet buffers are present on most line cards. Secondly, there is only one RLDRAM manufacturer left since the economical crisis in 2009. This poses a high economical risk, because discontinuation in RLDRAM production due to any reason leads to discontinuation in corresponding router production.

Summarizing, high-speed SRAMs can suffice the requirements of the descriptor memory, but no memory type is available today nor in foreseeable future that can suffice the requirements

[9]Ethernet is the dominant transmission technology in LANs and is predicted to be soon predominant in WANs too. Sommer *et al.* provide in [9] a survey of its fields of application.

of the packet memory in a high-speed packet buffer. However, many memories meet the requirements partly, e. g., DRAMs meet mostly the capacity requirement, while SRAMs meet mostly the access time requirement. Concluding, architectural concepts are required that utilize the strengths of both. Favorably, these should enable utilization of commodity DRAM (e. g., DDR3 SDRAM) to reduce system costs.

2.4 Basic Architectural Approaches

The previous Section faced high-speed packet buffer requirements and commercially available memories and concluded that there is a large discrepancy. This Section introduces basic approaches how to overcome limitations in memory parameters architecturally. The first Subsection gives an overview of the concepts while the following ones present explicit approaches. The last Subsection discusses and compares the approaches shown.

2.4.1 Overview

To solve the discrepancy between available and required memory parameters several architectural concepts have been proposed. Depending on the memory parameter that causes the shortcoming, different basic concepts can be applied. Table 2.3 gives an overview.

Shortcoming in parameter	**Architectural concept to overcome this**
bandwidth	parallelism
capacity	parallelism
access time	interleaving aggregation

Table 2.3: Basic architectural concepts to overcome limitations in memory parameters

To solve a shortcoming in memory **bandwidth**, simple parallelism is sufficient. The bandwidth increases linearly with the number of parallel memory devices. The same is true for the capacity, when disregarding the overhead for managing the increased **capacity**.

To solve the shortcoming in access time two principle concepts are applicable: interleaving and aggregation. *Interleaving* means that memory accesses are distributed to different memory devices, which are operated all independently. With n parallel memory devices the **access time** theoretically can be reduced by factor n, i. e., to T/n. The idea behind *aggregation* is that instead trying to decrease the access time the data blocks read/written to/from memory are increased to meet the access time, i. e., the larger the data block the lower the required access time. Therefore packet data is aggregated to larger data blocks. For example, in a system with a total memory bandwidth of $2R$ and a memory random access time T the block size has to be chosen $2R \cdot T$. In this system every T one block can be written to *or* read from memory.

Based on these concepts several architecture approaches for high-speed packet buffers exist. These can be classified into three basic groups: Parallel Memories, Hybrid Memory Architecture, and Hybrid Memory Architecture with Parallel Subsystems. All approaches focus on

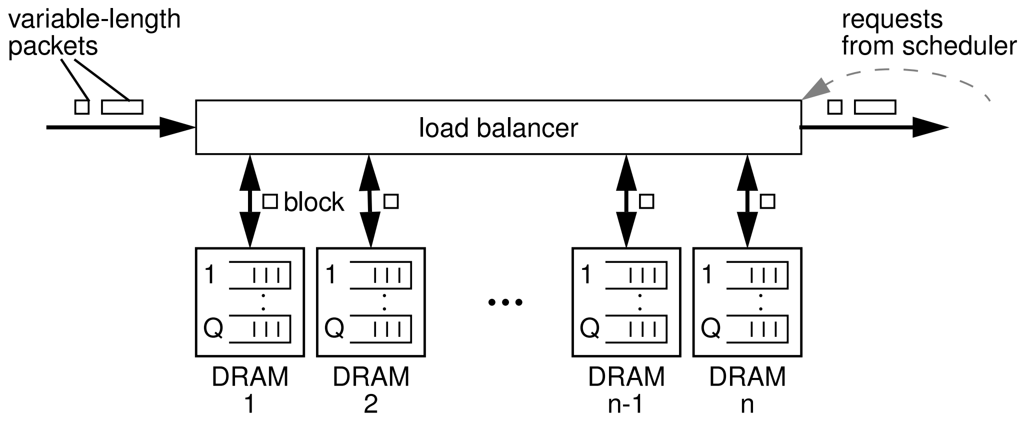


Figure 2.19: Basic architecture approach: Parallel Memories

achieving the required access time using DRAMs as this is the hardest challenge. Bandwidth and capacity is usually achieved by using more or wider DRAMs. The following Sections discus in detail their pros and cons. The base requirement for the following approaches is to support Q flow queues for buffering packets which are accessed only in FIFO manner.

2.4.2 Parallel Memories

Basic Idea

The basic idea of the *parallel memories approach* is, that with n parallel devices the access time theoretically can be reduced by factor n, i. e., to T/n, with T being the random access time of the memory device. Figure 2.19 shows the memory architecture of this approach.

Working Principle

From left to right in Figure 2.19 the *load balancer* segments incoming variable-length packets to constant-size blocks and distributes them to the independent DRAMs. Each DRAM maintains one queue per flow, while Q flows exist. When the external scheduler requests a packet, the load balancer requests the blocks of this packet from the corresponding DRAMs. In worst case, each DRAM access takes T, which is the random access time of the DRAM. To hide this long access time of an individual DRAM, the load balancer interleaves the accesses, i. e., there are many read and write accesses active concurrently.

Properties

The read accesses to the DRAMs are fixed as the location of the blocks of the requested packets explicitly determine them. To write the blocks of arriving packets, the load balancer can chose a DRAM that is currently non-busy. Hereby it distributes the blocks to the DRAMs in a way that they can be read later interleaved and concurrently. E. g., two blocks residing in the same DRAM cannot be read simultaneously, only back to back.

To achieve a deterministic bandwidth (100 % throughput) with such an architecture in the general case (packet arrivals and departures are unknown) a very large number of DRAMs is required. If $2g$ DRAMs are required to read *and* write simultaneously blocks of one data flow with line rate, then in this general case $2gQ$ independent DRAMs are required for Q different data flows.

However, there are also router architectures with deterministic access patterns. In these routers the packet departure times can be deterministically calculated before the packet is written to the packet buffer when best-effort routing is considered. Examples are the Parallel Shared Memory (PSM) router [71] and the load-balanced router architecture [72].

Under this constraint of knowing the packet departure time in advance $n = 3N - 1$ DRAMs are sufficient. Here $N = \frac{T}{T_{required}} = \frac{T}{b/R}$ is the ratio of the available and the required access time, b is the block size and R is the line rate at the packet buffer's input and output. An incoming block competes with $3N - 2$ other blocks for the DRAMs: $N - 1$ other arrivals, N departures, and $N - 1$ future departures which will be processed concurrently. Hence, if $n \geq 3N - 1$ the arriving block can find a non-busy DRAM. [73, 74] propose efficient implementations for such so called reservation-based packet buffer architectures.

The **Ping Pong buffering scheme** [75] is a special case of the parallel memory approach, where $n = 2$. In a high-speed packet buffer maintaining several FIFO queues there is only one read and one write process. The bandwidth required to write or read these blocks is at most the line rate each. This means, with two memories the write process always can chose the non-busy memory. Ping Pong has two big advantages. Firstly, it does not rely on knowing the packet departure times in advance. Secondly, it is applicable to nearly all other architecture approaches where it always reduces the access time by half. Its drawback is that in the worst case half of the memory capacity is wasted, i. e., blocks cannot be accepted even if there is free memory capacity. To illustrate this lets assume that the two memories A and B currently have the following state: A is full and B is nearly empty. While now reading blocks from memory B incoming blocks cannot be written to memory A as it is full.

2.4.3 Hybrid Memory Architecture

Basic Idea

The basic idea of the *hybrid memory architecture approach* (cf. Figure 2.20) can be summarized in three main points.

- Transfer large blocks to/from DRAM to match its *long* random access time.
- Aggregate user packets to blocks and vice versa utilizing SRAM.
- SRAM at the input and output interface of the packet buffer meets the required access time.

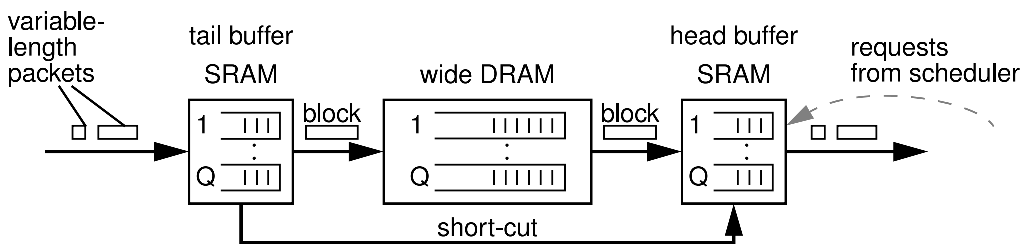


Figure 2.20: Basic architecture approach: Hybrid Memory Architecture

Working Principle

To simultaneously meet all requirements of the packet buffer the hybrid memory architecture combines the advantages of both, SRAM and DRAM. SRAM meets the random access time and DRAM the capacity requirement, while both meet the bandwidth requirement. For this approach it is assumed that an SRAM is available that meets the access time required by the system.

Figure 2.20 shows the basic architecture, which consists of three main components: tail buffer, head buffer, and wide DRAM. The tail buffer (SRAM) contains the tails of all flow queues, i. e., the packets that arrived last. The head buffer (SRAM) contains the heads of the flow queues, i. e., the packets that are going to leave soon. The DRAM in the middle holds all the other packets of the queues. It is realized by a sufficient number of DRAMs that are operated in unison, i. e., it corresponds to one wide logical DRAM. A Memory Management Algorithm (MMA) controls the data transfers between SRAM and DRAM. Its challenge is to never let the tail buffer overflow and to make sure that packets are always present in the head buffer when needed.

From left to right in Figure 2.20 the tail buffer aggregates per flow incoming packets to blocks. When enough data has arrived for one flow, i. e., a block is full, the tail-MMA initiates a data transfer to DRAM. Then this data block is transferred from tail buffer to the corresponding queue in DRAM. Similarly, in preparation for a packet departure, the head-MMA initiates block transfers from the corresponding queues in the DRAM to the head buffer.

A side effect of aggregation is that non-full blocks are never written to DRAM. Thus, for flows with light traffic it may happen that such a block is requested by the head part, but is not available in the DRAM. A *short-cut* path from tail to head buffer solves the problem.

Properties

To achieve a deterministic bandwidth the tail buffer is not allowed to overrun and the head buffer to underrun. The accordingly required tail and head buffer (SRAM) sizes are bounded and are a function of Q, R, and T.

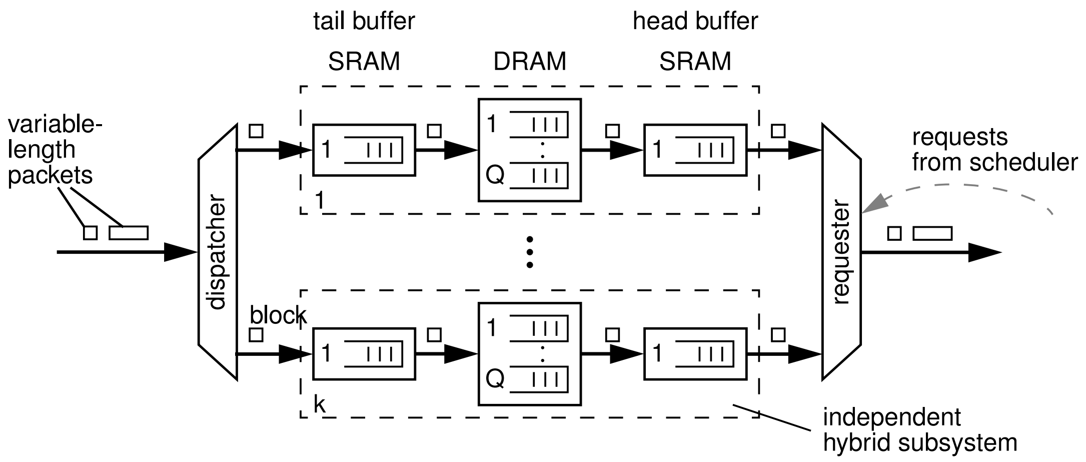


Figure 2.21: Basic architecture approach: Hybrid Memory Architecture with Parallel Subsystems

This approach enables a trade-off between the head buffer size and the read latency[10] of the packet buffer. With a read latency of zero, the packet buffer delivers each packet immediately after receiving the request. To achieve this, for each flow enough data has to be cached in the head buffer to not underrun until newly requested data from DRAM arrives. This has to hold for any access patterns. In contrast, if we accept a pipeline delay, i. e., a read latency > 0, a significantly smaller head buffer is required as the head-MMA starts requesting packets from DRAM only after it received a packet request. [76, 27] propose packet buffer architectures based on this approach.

This hybrid approach has two big advantages. Firstly, the aggregation to blocks eliminates fragmentation in SRAM and DRAM, i. e., no capacity overdimensioning is required. Further, as only full blocks are transferred to/from DRAM the DRAM bandwidth also requires no overdimensioning. Secondly, it delivers deterministic bandwidth without any assumption about the arrival or departure pattern.

However, it has also two main drawbacks. Firstly, the required sizes of the head and tail buffer are linearly tied to the parameters Q, R, and T. Accordingly, to support a large number of flows Q also a large tail and head buffer is necessary. Secondly, this approach cannot utilize the banks available in the DRAM. Consequently, without bank interleaving the DRAM bus is poorly utilized. Due to this the DRAM bus has to be dimensioned much wider to support the required bandwidth.

2.4.4 Hybrid Memory Architecture with Parallel Subsystems

Basic Idea

The idea of the *hybrid memory architecture approach with parallel subsystems* is to improve the plain parallel approach (Section 2.4.2) by utilizing SRAM like in the hybrid approach (Section 2.4.3). Figure 2.21 shows its architecture. The three key points thereby are:

- With n parallel devices the access time can be reduced theoretically by factor n, i. e., to T/n.
- Reduction of the number of parallel subsystems to a minimum (here k) that is sufficient to support the required access time.
- Utilization of SRAM to resolve temporal overbooking of a DRAM and for reordering.

Working Principle

Figure 2.21 shows the basic architecture that consists of k parallel subsystems that are operated interleaved. Each subsystem has a hybrid architecture similar to the one introduced in Section 2.4.3 consisting of three components: tail buffer, head buffer, and DRAM. Each tail buffer (SRAM) contains one DRAM queue. Here blocks are stored in the case when the DRAM is temporary overbooked, i. e., the DRAM receives more write requests than it can process. Each head buffer (SRAM) contains one DRAM queue which acts as a reorder buffer. The head buffer assures that independently of the load of the individual DRAMs the requested packets are delivered in-order.

In each subsystem the DRAM holds Q queues, i. e., all logical Q flow queues provided to the user are spread over all subsystems. An MMA controls the data transfers between SRAM and DRAM. Its challenge is to never let the tail buffer overflow and to make sure that packets are always present in the head buffer when needed.

From left to right in Figure 2.21 the dispatcher segments incoming variable-length packets to blocks and distributes them per flow equally to the individual subsystems. Only equal distribution allows to write and to read data of every flow with line rate, because only all subsystems together can support line rate. When a block is available in a tail buffer, the corresponding MMA initiates a data transfer to DRAM. Similarly, in each subsystem in preparation for a packet departure, the MMA initiates block transfers from the corresponding queues in the DRAM to the head buffer. [77, 78] propose architectures based on this approach.

[10]Read Latency refers to the time between issuing a read request to the packet buffer and receiving the corresponding data. This corresponds to the minimum delay that a packet buffer introduces to every packet.

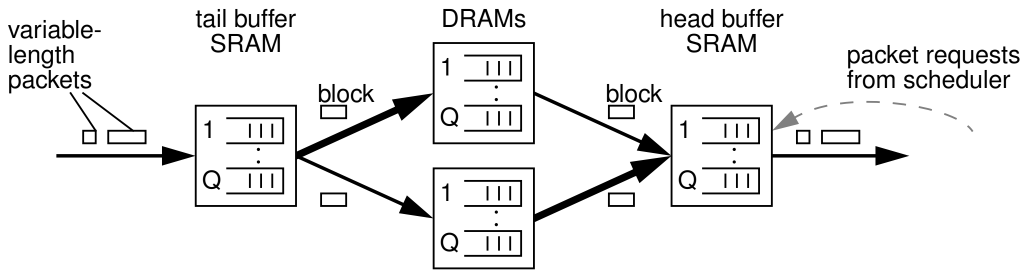


Figure 2.22: Hybrid Memory Architecture with Ping Pong Approach; This example illustrates a time instant, where the MMA writes a block to the upper DRAM and simultaneously reads a block from the lower DRAM.

Properties

To achieve a deterministic bandwidth no tail buffer is allowed to overrun and no head buffer to underrun. The accordingly required tail and head buffer (SRAM) sizes are bounded and are a function of Q, R, and T.

The DRAMs of the individual subsystems can be implemented as DRAM banks, e. g., if $k = 8$ subsystems are required then the corresponding DRAMs can be implemented with two DRAMs with 4 banks each. Consequently, interleaved operation of DRAM banks greatly increases the DRAM bus utilization.

This hybrid approach with parallel subsystems has three main advantages. Firstly, it supports banking, i. e., the interleaved access to DRAM banks. The increased DRAM bus utilization leads to more narrow DRAM bus what lowers costs. Secondly, due to the individual subsystems distributed implementation is possible. Thirdly, it delivers deterministic bandwidth without any assumption about the arrival or departure pattern.

Its main drawback is that the required sizes of the head and tail buffer are linearly tied to the parameters Q, R and T. Accordingly, to support a large number of flows Q also a large tail and head buffer is necessary.

2.4.5 Discussion

The previous Sections presented three architectural approaches to overcome the shortcomings in memory parameters. These are discussed in the following.

To provide deterministic bandwidth in the first approach (*parallel memories*, cf. Section 2.4.2) the packet buffer needs to know the packet departure times before packets enter. This is not the case in general high-speed routers supporting QoS. Therefore, this approach is disregarded in the following.

In contrast, the two *hybrid approaches* make no assumption about the traffic arrival and departure pattern. Beside this, the hybrid approaches scale towards higher line rates and are in principle resource efficient in terms of DRAM capacity and bandwidth.

Their main drawback is that the required SRAM size to realize head and tail buffer scales linearly with number of supported flows Q. To alleviate this shortcoming the Ping Pong approach [75] described in Section 2.4.2 can be used. Therefore each individual DRAM is divided to a DRAM pair, i. e., each DRAM of the DRAM pair has half of the original bus width and accordingly delivers half of the bandwidth. This step does not change the total DRAM capacity or bandwidth. The random access time of a DRAM pair is only $\frac{T}{2}$, i. e., compared to a single DRAM it is cut by half. Figure 2.22 shows the resulting architecture. Due to the linear connection between head and tail buffer sizes to the random access time of the DRAM head and tail buffer sizes are also cut by half. The price for this SRAM size reduction is that in worst case only 50 % of the DRAM capacity can be used (cf. Section 2.4.2). Alisafaee *et al.* propose in [79] a Ping Pong based packet buffer architecture that tries to alleviate the problem of DRAM capacity usage.

Another approach to reduce the SRAM size is to simply underdimension it. This leads inevitably to occasional loss of blocks in the packet buffer. The external effects are incoming packets that are not buffered, requested but not delivered packets, eventually non-constant read latencies and of course only a statistical bandwidth guarantee. This can be acceptable depending on the system. The authors in [77, 80, 81] analyze the effect of SRAM underdimensioning on the drop probability in hybrid packet buffers.

2.5 Survey of Hybrid Memory Architectures

In this Section hybrid memory architectures for packet buffers are discussed in more detail. According to the scope of this thesis, here only architectures with deterministic bandwidth are considered. Deterministic bandwidth does not only mean 100 % throughput, but also no loss, strict in-order delivery, and constant read latency. These properties imply a simple SRAM-like packet buffer interface.

The first Subsection introduces important metrics to allow quantified comparison. Following Subsections present in detail hybrid memory architectures known in literature. Functionality of these complies with the packet buffer definition from Section 2.2.2, except the fact that they do not provide queuing information to an external scheduler. The latter task can be implemented completely independently. To simplify reading the term packet buffer will be used for the architectures presented in the following as well. A discussion on their pros and cons concludes the Section.

2.5.1 Metrics

To compare packet buffer architectures we need to quantify their properties. The approach used here is requirements-driven, i. e., the resource requirements for a specific number of supported flow queues (Q) at line rate (R) are compared. An alternative approach would be to take a fixed amount of resources and derive the amount of flow queues (Q) supported by the individual architectures at line rate (R). The individual metrics can be classified into two groups: architecture- and realization-related.

Architecture-related metrics include all resource requirements and properties that originate from the underlying architecture. These are explicitly the *tail buffer size*, *head buffer size*, and the *read latency* introduced.

Realization-related metrics include the real resource requirement of an implementation and the achieved performance. Explicit examples are the real required SRAM size, the DRAM I/O pin count, the DRAM bus utilization, and the achievable performance, e. g., clock frequency or supported line rate.

As no relevant realization-related information is available in literature, the comparison focuses on the architecture-related metrics. However, if possible, discussions in the following Sections will cover also realization-related aspects. The architecture-related metrics are defined in the following:

Definition 2.1. Tail Buffer size, S_{tail} *This is the tail buffer size (SRAM) that is required in the worst case. An tail buffer dimensioned accordingly will never overflow.*

Definition 2.2. Head Buffer size, S_{head} *This is the head buffer size (SRAM) that is required in the worst case. An head buffer dimensioned accordingly will never underrun.*

Definition 2.3. Read Latency, L *This is the time between issuing a read request to the packet buffer and receiving the corresponding data. Due to the deterministic behavior the read latency is constant. It corresponds to the minimum delay that a packet buffer introduces to every packet.*

2.5.2 Related Work

Three basically different hybrid memory architectures have been proposed in literature that provide deterministic bandwidth. These are introduced in the following.

2.5.2.1 Hybrid SRAM/DRAM System (HSD)

Sundar Iyer *et al.* were the first to introduce a hybrid packet buffer architecture with deterministic bandwidth guarantee and give strict bounds for the head and tail buffer size. They first published the architecture in [76] and detailed it in [82, 45].

Architecture

The architecture corresponds to the *hybrid memory architecture* approach presented in Section 2.4.3. In the following it will be called the *Hybrid SRAM/DRAM System (HSD)*. Figure 2.23 shows its architecture. It maintains Q flow queues (one per flow) and stores the heads and tails of all queues in the corresponding buffer. It stores the remaining part in DRAM. The DRAM is realized by a sufficient number of DRAMs that are operated in unison, i. e., it corresponds to one wide logical DRAM.

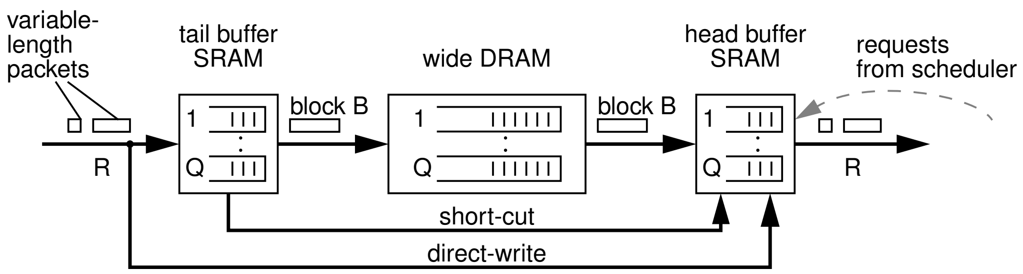


Figure 2.23: Hybrid SRAM/DRAM System (HSD) of Iyer *et al.*

Working Principle

The tail buffer aggregates packets to blocks while the head buffer de-aggregates them accordingly. The single DRAM has a random access time of T and is accessed alternatingly by head and tail buffer. This means both, head and tail buffer, can access the DRAM once every $2T$. Data is transferred to and from DRAM always in blocks of size $B = 2TR$. As during $2T$ no more than $2TR$ of data can arrive and no more than $2TR$ of data can depart, the system is stable.

As soon as for a flow B byte of data have arrived in the tail buffer, the Tail Memory Management Algorithm (tail-MMA) initiates the transfer of B byte to the corresponding queue in DRAM. Similarly, in preparation for a packet departure, the Head Memory Management Algorithm (head-MMA) initiates a data transfer of B byte from the corresponding queue in DRAM to the head buffer.

It may happen that several blocks get full in the tail buffer in a short time period. The tail-MMA queues the blocks and transfers them one by one to the DRAM. The head-MMA requests a block that is not yet written to the DRAM directly from the tail buffer. These blocks are then delivered via the short-cut path.

The head-MMAs proposed by the authors require that the first $B-1$ byte data of every flow are directly written to the head buffer. This is done via the direct write path from the packet buffer input to the head buffer.

Properties

The authors prove, that the required tail buffer size is

$$S_{tail-HSD} = QB$$

when using dynamic memory allocation in the tail buffer.

The head buffer size and the read latency depend on the implemented head-MMA. The smallest possible head buffer size

$$S_{head-HSD-ECQF} = QB(1 - \frac{1\,\text{byte}}{B})$$

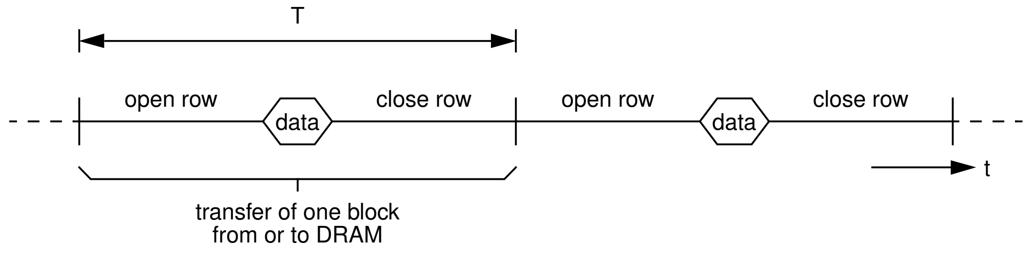


Figure 2.24: DRAM access in HSD

is achieved with the Earliest Critical Queue First (ECQF) head-MMA using dynamic memory allocation if a read latency of

$$L_{HSD-ECQF} = \frac{QB(1-\frac{1\,\text{byte}}{B})+1\,\text{byte}}{R}$$

is accepted. With the Most Deficit Queue First (MDQF) head-MMA the head buffer delivers every requested packet immediately upon a request, i. e., the read latency is zero. In this case the required head buffer size is

$$S_{head-HSD-MDQF} = QB(3+\ln Q)$$

while the head buffer is allocated statically to the individual queues. However, the MDQF head-MMA is only practical for small number queues [45].

The tail buffer itself introduces no delay due to the short-cut path, i. e., it does not increase the read latency. Finally, a DRAM bandwidth of $2R$ is sufficient, as only full blocks are transferred to and from DRAM.

Discussion

The HSD aggregates packets to blocks. Aggregation eliminates fragmentation in SRAM and DRAM, i. e., no capacity overdimensioning is required.

The HSD assumes one wide DRAM that can be accessed once per random access time and does not utilize bank interleaving. This leads to a low DRAM bus utilization because the DRAM is most of the time occupied with opening and closing the rows. Figure 2.24 illustrates this. For a numerical example a DRAM of the type DDR3-1600K SDRAM [29] is assumed. Currently, this is one of the DRAMs with the highest bandwidth. The author uses an approximate calculation to derive the random access time T of this DRAM device. From this it results, that for a read/write ratio of 1 and transferring one burst per access, the mean random access time is $T = 53.1$ ns. The mean value is required, because a read and a write access have different access times. This means, ignoring other effects[11] one read and one write access can be performed in $2T = 106.2$ ns, while the read access takes 48.75 ns and the write access 57.45 ns. Transferring

[11]Using DRAMs, there are large penalties when changing the access type, i. e., from read to write or vice versa [29]. To minimize the effect of these penalties the DRAM controller can accumulate read and write access and then process them as batch [31], i. e., the access pattern `RWRWRWRW` is transformed to `RRRRWWWW` at the cost of the latency and buffer that is required for accumulation.

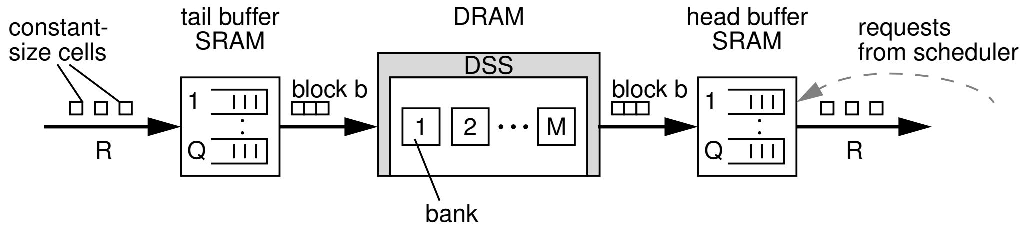


Figure 2.25: Conflict-Free DRAM System (CFDS) of Garcia *et al.*

one burst of 8 words takes 4 clock cycles of 1.25 ns each, i. e., 5 ns. This leads to a DRAM bus utilization of $\frac{5}{53.1} \approx 10\,\%$. Utilization can be increased by transferring more than one burst per access. With i bursts per access the mean random access time is $T_i = (53.1 + (i-1)\cdot 5)$ ns, and consequently, the bus utilization is $\frac{i\cdot 5}{53.1+(i-1)\cdot 5}$. However, increasing the random access time increases also the block size B and, thus, the head and tail buffer size.

Finally, the proposed head-MMAs require an additional direct write path to the head buffer. Thus the head buffer needs a minimum bandwidth of $3R$.

2.5.2.2 Conflict-Free DRAM System (CFDS)

Garcia *et al.* propose a hybrid architecture that exploits banking to reduce tail and head buffer size. They first published the idea in [83] and detailed it in [84, 27, 85]. It exploits the fact, that with n banks the random access time T of a system theoretically reduces to T/n, as interleaving of bank accesses ideally increases the total number of accesses per time unit by factor n. A shorter access time leads to a smaller block size which leads to a smaller head and tail buffer size and a shorter read latency.

Architecture

The architecture is similar to the HSD architecture of Iyer *et al.* presented in Section 2.5.2.1. The main difference is that here the authors utilize the banks in the DRAM to improve the architecture. The authors call their system the *Conflict-Free DRAM System (CFDS)* as it accesses the individual banks in a way that no bank conflicts occur at all. Figure 2.25 shows its architecture.

It maintains Q flow queues (one per flow) and stores the heads and tails of all queues in the corresponding buffer. It stores the remaining part in DRAM spread over all banks. The DRAM is realized by a sufficient number of DRAM devices that are operated in unison, i. e., it corresponds to one wide logical DRAM. The authors assume this logical DRAM to have up to $M = 512$ banks, i. e., each of its DRAM devices has to have M banks. The DRAM Scheduling Subsystem (DSS) component hides the DRAM bank organization from the former MMA.

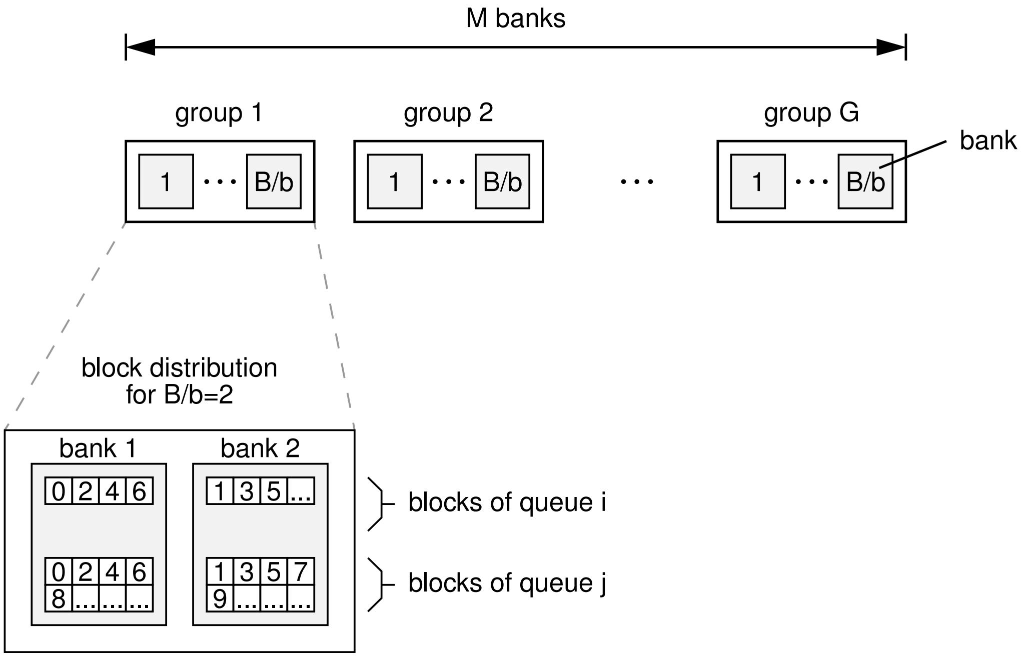


Figure 2.26: Bank organization of the CFDS; The example on the lower left shows the per queue round robin distribution of the blocks to the banks using $B/b = 2$ (Source [27])

Working Principle

The authors assume their packet buffer receiving and delivering constant-size data units of 64 byte which they call cells. A preceding component generates the cells by segmenting variable-length packets.

The tail buffer aggregates per flow an integer number of cells to a block while the head buffer de-aggregates them accordingly. The single DRAM has an imaginary random access time of T_{imag} due to the use of banking and is accessed alternately by head and tail buffer. This means both, head and tail buffer, can access the DRAM once every $2T_{imag}$. Data is transferred to and from DRAM always in blocks of size $b = 2T_{imag}R$. b is an integer number of cells, i. e., T_{imag} has to be dimensioned accordingly.

As soon as for a flow b cells arrived in the tail buffer, the tail-MMA initiates the transfer of b cells to the corresponding queue in DRAM. Similarly, in preparation for a packet departure, the head-MMA initiates a data transfer of b cells from the corresponding queue in DRAM to the head buffer.

The organization of the memory banks in DRAM is as follows. The total number of M banks is organized as $G = M/(B/b)$ groups of B/b banks per group (cf. Figure 2.26). $B = 2TR$ is the block size in the case when bank interleaving is not used with T being the random access time of the DRAM. Each group stores blocks of Q/G queues. With bank interleaving the block size b is reduced compared to B by the factor $\frac{B}{b} = \frac{T}{T_{imag}}$. Consequently, to read or write blocks b of a flow with line rate, $\frac{B}{b}$ more accesses are necessary compared to the case without bank interleaving and block size B. Due to this reason, in order to avoid bank conflicts, the blocks in each queue are distributed in a round robin manner among all $\frac{B}{b}$ banks of the group to which the queue is assigned. $\frac{B}{b}$ has a hard a technological upper bound as the DRAM bus frequency and the timing constraints limit the number of accesses that can be performed during the time T (cf. Section 2.3.3).

The DSS hides the bank organization from the tail- and head-MMA, which operate under the illusion that the DRAM access time is T_{imag}. This is the main difference to the HSD architecture of Iyer *et al.* (cf. Section 2.5.2.1). To resolve the unavoidable bank conflicts the DSS has a request buffer where it queues read and write requests which it processes out-of-order. This means the requests experience a delay here. Accordingly, the DSS also contains a reorder buffer to re-sequence the read blocks before they are delivered to the head buffer. With this out-of-order processing the DSS can resolve all bank conflicts what makes the system to appear from outside conflict-free.

Properties

Garcia *et al.* analyze only the head side of the architecture. The authors prove that with the described bank organization the delay introduced by the DSS is bounded. Accordingly, the required reorder buffer is also bounded in size.

The total required amount of SRAM on the head side is composed of two parts: the amount required due to the used head-MMA and the amount due to the DSS. Using the ECQF head-MMA from the HSD architecture, which minimizes the head buffer size, the total SRAM size required on head side is

$$S_{head-CFDS-ECQF} = \underbrace{Qb}_{\text{due to head-MMA}} + \underbrace{(2Q/G)-1)(B/b-1)b}_{\text{due to DSS}}$$

The introduced read latency is analog to the SRAM size in the head composed of two parts

$$L_{CFDS-ECQF} = \underbrace{\frac{Qb}{R}}_{\text{due to head-MMA}} + \underbrace{\frac{(2Q/G)-1)(B/b-1)2b}{R}}_{\text{due to DSS}}$$

The MDQF head-MMA from the HSD architecture, which reduces the read latency, is also applicable here. Utilizing the MDQF head-MMA, total SRAM size and read latency just change, compared to the ECQF head-MMA, in the part introduced by the head-MMA.

There is an optimal value b for a given CFDS implementation that minimizes the total required SRAM size. The reason for this is the trade-off between the head buffer size on the one hand, which is proportional to b, and the reorder buffer size in the DSS on the other hand, which is proportional to $\frac{1}{b}$.

The CFDS has the drawback of DRAM fragmentation. The reason for this is, that it statically assigns queues to the G memory groups. For example, if the banks associated to a group are full, no more data for the corresponding queues can be accepted, even if the rest of the DRAM is empty.

To alleviate the fragmentation problem the authors introduce in [27] a queue renaming mechanism. Therefore they differentiate between Q logical queues visible at the packet buffer interface and P physical queues that are maintained by the architecture. At the packet buffer input, the logical queues are mapped (renamed) to physical queues. A renaming buffer for each logical queue stores these mappings. Lets assume, that the logical queue q_1 is mapped to the physical queue p_3. If a new arriving cell for q_1 finds that the DRAM associated to the group of p_3 is full, then q_1 is additionally mapped to a new physical queue in a different group which can offer free DRAM. By doing this, blocks from a logical queue can reside in more than one memory group and potentially can occupy the whole DRAM. Each physical queue is mapped exclusively to one logical queue. With this queue renaming mechanism all results remain valid, when Q is replaced by P in the formulas.

In order to work it has to be $P > Q$, what leads to an increase in SRAM size and read latency. Further, DRAM memory fragmentation can still arise for specific traffic patterns. Fragmentation occurs, when all physical queues are mapped to logical queues, e. g., this can happen when there are logical queues with blocks scattered over many physical queues. New renamings are now not possible any more, even if the DRAM is nearly empty.

Discussion

Garcia *et al.* propose a system, that improves the HSD architecture by exploiting banking. They thereby improve both, overall SRAM requirement and read latency. Improvements increase with the number of available DRAM banks. Therefore, the authors assume DRAMs with several hundred banks, e. g., $M = 512$. However, to the best of our knowledge such DRAMs do not exist now and in foreseeable future.

Nevertheless, for a system with $R = 100$ Gbps and DDR3 SDRAM ($M = 8$ banks) the SRAM on head side is reduced compared to the HSD architecture. Comparison assumes the ECQF head-MMA. To find the optimal configuration for this CFDS Table 2.4 shows all possible configurations when using a standard DRAM with $M = 8$ banks (cf. Table 2.2). Accordingly, the parameters $G = 4$ groups with $B/b = 2$ banks per group lead to the minimum SRAM requirement of $\frac{3}{4}QB$ byte on head side. Compared to the HSD architecture with QB byte on head side this is a saving of 25 %.

configuration		head buffer	reorder buffer	total SRAM
$\frac{B}{b}$	groups G	size [byte]	size in DSS [byte]	in head part [byte]
1	8	QB	0	QB
2	4	$\frac{1}{2}QB$	$\frac{1}{4}QB$	$\frac{3}{4}QB$
4	2	$\frac{1}{4}QB$	$\frac{3}{4}QB$	QB
8	1	$\frac{1}{8}QB$	$\frac{14}{8}QB$	$\frac{15}{8}QB$

Table 2.4: Possible CFDS configurations utilizing a DRAM with $M = 8$ banks

Due to the systematic use of bank interleaving the CFDS utilizes the DRAM bus efficiently. This is highly cost and resource saving. The price is the additional effort for DRAM bank management performed by the DSS.

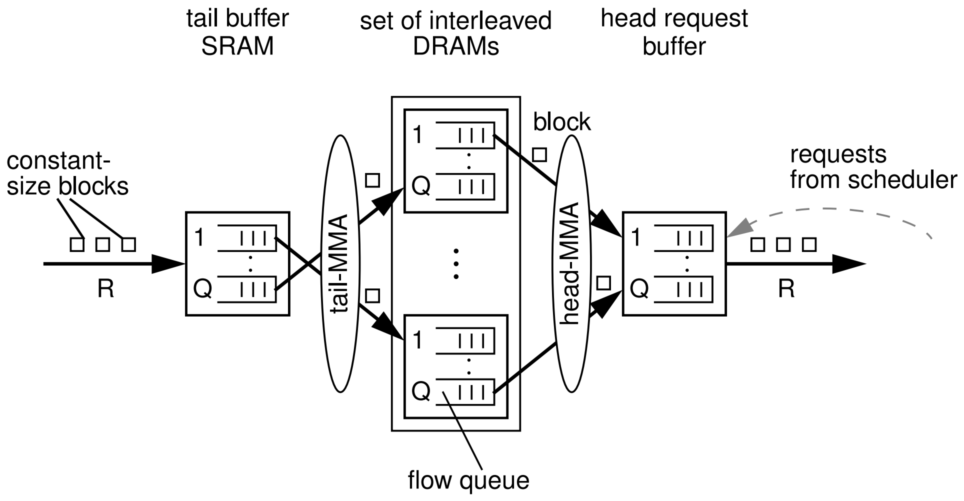


Figure 2.27: Hybrid memory architecture with interleaved DRAMs of Wang *et al.*

The authors never mention the presence of a short-cut from tail to head buffer like it is available in the HSD architecture (cf. Section 2.5.2.1). However, their formula for the latency L does not account for any latency introduced by the tail part. Without short-cut the latency L doubles due to symmetry.

The authors claim to use the same ECQF MMA as in the HSD architecture. The ECQF MMA requires a direct write path from packet buffer input to the head buffer. The authors never mention this. Consequently, the head buffer needs a minimum bandwidth of $3R$.

The CFDS suffers from external and internal fragmentation. The authors alleviate the external fragmentation, which affects the usable DRAM capacity, by their queue renaming mechanism, but actually cannot eliminate it. The main price for this is, that they have to maintain P instead of Q queues, with $P > Q$. This increases the required SRAM size as well as the read latency. Further, the required renaming buffers (Q pieces) add costs too.

Internal fragmentation occurs because the authors assume cells arriving at and departing from their packet buffer. As cells are segmented packets this leads to the 65-byte-problem, i. e., there is a worst-case packet size that leads to two cells where the second contains just one byte. This means, that all internal (SRAM) and external (DRAM) memory capacity is double as large as without the 65-byte-problem. In other words, the system is dimensioned for a gross input/output data rate instead for the net data rate R, which is just half of that.

2.5.2.3 *Parallel Hybrid SRAM/DRAM System (PHSD)*

Wang *et al.* propose a hybrid memory architecture that exploits parallelism. The authors target with their architecture a better scalability to larger packet buffer capacities and a short read latency.

The authors first published their idea in [86, 87]. Here they introduce a hybrid memory architecture with interleaved DRAMs and single SRAM in the tail and head (cf. Figure 2.27). They study the properties of the tail-MMA, which bases on a matching algorithm, and the tail buffer.

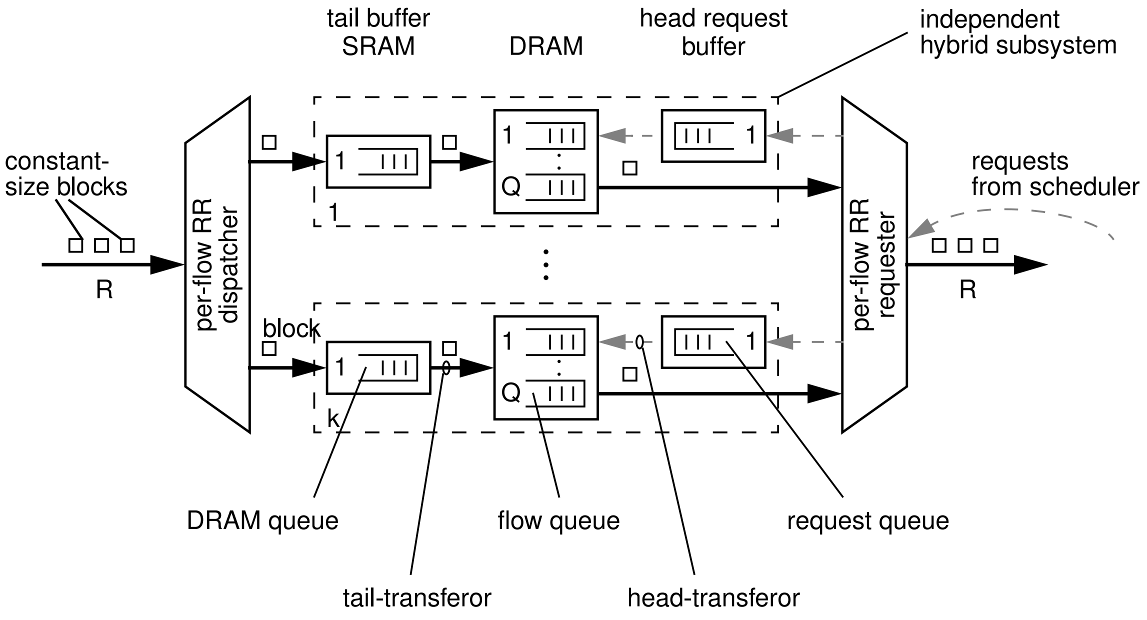


Figure 2.28: Parallel Hybrid SRAM/DRAM System (PHSD) of Wang *et al.*; The reader is noticed, that the head side of the architecture is drawn differently compared to the original figure of Wang *et al.*, because the original figure does not differentiate between blocks and block requests.

They conclude that the tail-MMA implementation complexity is high. In [78, 88, 89] Wang *et al.* propose an improved architecture, which they call *Parallel Hybrid SRAM/DRAM System (PHSD)* as it is hybrid and features parallelism. This Section introduces the PHSD in detail.

Architecture

The PHSD bases on the *hybrid memory architecture with parallel subsystems* approach presented in Section 2.4.4. The main difference is that the PHSD has no head buffer.

Figure 2.28 shows its architecture. It consists of *k* hybrid subsystems that are arranged in parallel. Each subsystem consists of 3 components: tail buffer, DRAM and head request buffer. Each tail buffer (SRAM) contains one DRAM queue. This queue stores blocks in the case when the DRAM is temporary overbooked, i. e., the DRAM receives more write request than it can process. In each subsystem the DRAM holds *Q* queues, i. e., the logical *Q* flow queues provided to the user are all spread over all subsystems. Each head request buffer (SRAM) contains one request queue. This queue stores block requests in the case when the DRAM is temporary overbooked, i. e., it receives more read requests than it can process.

Two MMAs control the data transfers between SRAM and DRAM: the head-MMA and the tail-MMA. The tail-MMA consists of two components: the dispatcher and the tail-transferor. The dispatcher distributes incoming blocks to the individual subsystems while the tail-transferor initiates the block transfers from tail buffers to DRAMs. Analogously, the head-MMA also consists of two components: the requester and the head-transferor. The requester distributes incoming block requests to the individual subsystems while the head-transferor initiates the transfer of block request from head request buffers to DRAMs.

Working Principle

The authors assume their packet buffer receiving and delivering constant-size data blocks. A preceding component generates these blocks by segmenting variable-length packets.

To minimize waiting times, incoming blocks are not aggregated to larger units. This has two consequences. Firstly, the access granularity of SRAM and DRAM is one block. Secondly, the block size defines the minimal number of parallel subsystems (k_{min}) required for the system to be stable. k_{min} is the ratio of the block transmission time at line rate R and twice the DRAM random access time because head and tail side can access the DRAM each once every $2T$:

$$k_{min} = \frac{2T}{t_{block}} = \frac{2T}{\frac{b}{R}} = \frac{2TR}{b} = \frac{B}{b}$$

With this dimensioning each subsystem provides $1/k_{min}$ of the total required bandwidth.

To be able to read and write blocks of a flow with line rate its blocks have to be uniformly distributed to all subsystems, as only all together can deliver line rate. Therefore, the dispatcher distributes the blocks *per flow* in a round robin manner to the subsystems.

From left to right in Figure 2.28 the dispatcher distributes the incoming blocks to the individual subsystems. As soon as a tail buffer is non-empty the tail-transferor initiates the transfer of the queue head to the corresponding queue in the DRAM. Similarly, from right to left in Figure 2.28 the requester distributes incoming block requests and the head-transferor initiates the transfer of the block requests to the DRAM. Blocks delivered by the DRAMs are directly transferred to the packet buffer output.

Properties

Like targeted by the authors, the PHSD scales to very large capacities by simply increasing the number of parallel subsystems k. For the system to be stable each subsystem has to still provide $1/k_{min}$ of the total required bandwidth. Inversely this means, that when $k > k_{min}$ the system is overdimensioned.

The authors prove, that the total tail buffer size of all subsystems is

$$S_{tail-PHSD} = Qbk(1 - \frac{1}{k})$$

The factor $-\frac{1}{k}$ accounts for the fact, that due to the absence of aggregation, blocks arriving to an empty DRAM queue can be immediately transferred to DRAM, i. e., less blocks can accumulate per DRAM queue. Disregarding the factor $-\frac{1}{k}$ and $k = k_{min}$ the tail buffer size is equal to that in the HSD architecture: $Qkb = Qk\frac{2TR}{k} = QB$. In other words, the blocks in the PHSD are k times

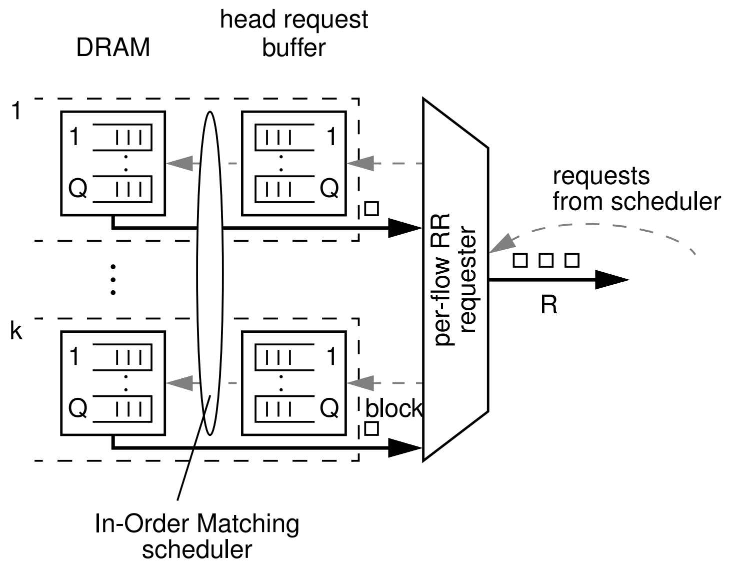


Figure 2.29: Head side of PHSD with in-order matching scheduler; The reader is noticed, that the architecture is drawn differently compared to the original figure of Wang *et al.*, because the original figure does not differentiate between blocks and block requests.

smaller compared to the HSD, but the PHSD also has k times more tail buffers. Consequently, choosing $k > k_{min}$ leads also to a proportional increase in tail buffer size ($S_{tail-PHSD}$).

Block requests are more than an order of magnitude smaller than blocks. Consequently, the size of the head request buffer, which is also realized with SRAM, can be neglected.

The read latency is introduced by waiting block requests in the head request buffer. Its value is non-constant due to the absence of a reorder buffer, i. e., head buffer. The authors prove, that its value is bounded to

$$L_{PHSD-max} = \frac{Qbk_{min}(1-\frac{1}{k})}{R} = \frac{QB(1-\frac{1}{k})}{R}$$

Analogously to the tail side, disregarding the factor $-\frac{1}{k}$ the maximum read latency introduced by the PHSD is equal to the constant read latency of the HSD architecture.

The largest drawback of the PHSD is that blocks requested by the scheduler are delivered out-of-order. Out-of-order delivery implies additionally a non-constant read latency as mentioned before.

In [88] the authors modify their architecture to alleviate the out-of-order problem algorithmically. The main change is that the k head transferors no longer work independently, but are coordinated by an In-Order Matching (IOM) scheduler. Figure 2.29 shows the head side of the new PHSD. The IOM scheduler implements a matching algorithm that is operated in rounds of $2T$. Each round it selects from each head request buffer one block request considering the constraint that the blocks of each flow are delivered in-order. To be able to select block request of any flow each head request buffer maintains Q request queues instead of the one before.

However, the achieved per-flow in-order delivery is still out-of-order compared to the requests sent by the scheduler and the read latency is also still variable. Proposed matching algorithms that are practical require an overdimensioning in the number of parallel subsystems k.

The architecture has further advantages. Firstly, it can be implemented distributed as the individual subsystems are independent. Secondly, due to the absence of a head buffer it requires in total only half of the SRAM size compared to the HSD architecture. Finally, the mean read latency is significantly below $L_{PHSD-max}$.

Discussion

Wang *et al.* propose a packet buffer architecture that allows distributed implementation. Due to the similarity to the HSD architecture (cf. Section 2.5.2.1) the properties are also similar. The authors omit the head buffer to save resources at a very high price: only per-flow in-order delivery, variable read latency, and mandatory overdimensioning in the number of subsystems k. If this non-deterministic behavior is acceptable depends on the targeted system. Nevertheless, the tail side of the architecture behaves deterministically.

The DRAM bus utilization is equally poor as in the HSD since no banking is used. Even if not regarded by the authors, the individual DRAMs in the subsystems can be replaced by individual DRAM banks increasing the bus utilization. Using banks would reduce the total DRAM capacity of the system. However, this is contrary to the authors' intention to achieve large packet buffer capacities.

The authors never mention the presence of a short-cut from tail buffer to head part like it is available in the HSD architecture. However, their formula for the latency $L_{PHSD-max}$ does not account for any latency introduced by the tail part. Without a short-cut the maximum latency $L_{PHSD-max}$ doubles due to symmetry.

Finally, the PHSD suffers from internal fragmentation. This occurs because the authors assume constant-size blocks, arriving at and departing from their packet buffer. As these blocks are segmented packets this leads to the 65-byte-problem, i. e., there is a worst-case packet size that leads to two cells where the second contains just one byte. This means, that all internal (SRAM) and external (DRAM) memory capacity is double as large as without the 65-byte-problem. In other words, the system is dimensioned for a gross input/output data rate instead for the net data rate R, which is just half of that.

2.5.2.4 Discussion

This Section discusses the properties of the three architectures presented in detail: HSD, CFDS, and PHSD. Table 2.5 summarizes their main metrics, i. e., formulas for tail and head buffer size and read latency.

For comparison, one should consider, that to alleviate external fragmentation, the CFDS has to maintain P instead of Q queues, with $P > Q$. This increases the tail and head buffers size as well as the read latency of the CFDS (cf. Section 2.5.2.2).

	tail buffer size S_{tail}	head buffer size S_{head}	read latency L
HSD (ECQF)	QB	QB	$\frac{QB}{R}$
CFDS (ECQF)	not analyzed	$Qb+ (2Q/G)-1)(B/b-1)b$	$\frac{Qb}{R}+ \frac{(2Q/G)-1)(B/b-1)2b}{R}$
PHSD	Qbk	0	$\frac{QB}{R}$ (max. latency)

Table 2.5: Summary of the related work's main metrics; For simpler comparison, the metrics of HSD and PHSD are cleared from effects that come from different assumptions on arriving data granularity, i. e., they are slightly larger then the original metrics. From the metrics of the CFDS this effect is not removable.

Disregarding the 65-byte-problem, all metrics of all architectures are quite similar. Tail buffer size of HSD and PHSD are even equal. The CFDS reduces the head buffer size compared to the HSD by 25 % when utilizing DDR3-SDRAM (cf. example in Section 2.5.2.2). All metrics are proportional to the number of flows (Q) and due to the relationship $B = 2TR$ also to the line rate (R) and DRAM random access time (T). Since R and T are technologically given parameters and the SRAM sizes are limited today to a few Mbyte, the number supported flows in a deterministic packet buffer is moderate.

Considering the 65-byte-problem that causes internal fragmentation, the metrics of the architectures differ largely. Explicitly, CFDS and PHSD suffer from internal fragmentation, i. e. these architectures have to support in worst case an incoming line rate of $R' = 2R$. Consequently, for a deterministic packet buffer this means, that all internal (SRAM) and external (DRAM) memory capacity and bandwidth doubles. Concluding, under this constraint the HSD is nearly double as resource efficient according to the metrics than CFDS and PHSD.

	no internal fragmentation (SRAM and DRAM)	no external fragmentation (DRAM only)	DRAM banking	in-order delivery	constant read latency
HSD	✓	✓	–	✓	✓
CFDS	–	–	✓	✓	✓
PHSD	–	✓	–	per-flow	–

Table 2.6: Summary of the related works' properties

Table 2.6 summarizes further important properties of the three architectures: internal and external fragmentation, banking, in-order delivery, and read latency.

The required *DRAM bandwidth* and the *DRAM data bus utilization* of a system directly translate to the required DRAM pin count, which is a large cost factor. The required DRAM bandwidth is minimal, when there is no internal fragmentation. DRAM data bus utilization is high, when the system uses banking. However, no presented system features both.

The PHSD supports only per-flow in-order delivery and has consequently a variable read latency. If this is acceptable depends on the target system. Nevertheless, its tail side operates deterministically.

Summarizing, the presented architectures have all strengths but also weaknesses, especially concerning the efficient use of resources. These weaknesses not only increase system costs but also limit their feasibility considering future line rates.

A desirable future proof architecture optimally utilizes its resources and so requires no overdimensioning at all, i. e., it suffers from no fragmentation and it uses DRAM banking. It also requires a relatively small tail and head buffer size to be cost efficient.

3 A Novel Hybrid Memory Architecture for High-Speed Packet Buffers

The core of the Internet is built of high-speed routers and switches containing up to hundred and more ports. Temporary unbalanced traffic in a router and the presence of different CoS require high-speed packet buffers to hold packets during times of congestion.

Chapter 2 introduced the functional and technological requirements a packet buffer has to fulfill. It also introduced performance data and other properties of current and announced memory devices. Facing these with the technological requirements of a high-speed packet buffer shows, that at a line rate of $R = 10\,\text{Gbps}$ only one memory device and at $R = 100\,\text{Gbps}$ no memory device can fulfill all requirements simultaneously. This performance gap is going to increase in future as line rates increase much faster than memory performance.

A hybrid memory architecture can overcome the technological limitations by combining the strengths of the two major memory technologies: short random access time of SRAM and large capacity of DRAM. However, as discussed in Chapter 2, architecture proposals in literature that provide deterministic bandwidth suffer from high memory resource requirements.

This Chapter introduces a novel hybrid memory architecture for packet buffers delivering deterministic bandwidth. Its main advantage over current solutions is its significantly lower resource requirement. Among others, one main achievement is the large reduction in SRAM size, which is usually limiting feasibility. Consequently, this not only leads to a better scalability but also makes its implementation more cost efficient. The author presented and discussed these results on an international conference on router architectures [1].

This Chapter is organized as follows. Section 3.1 gives a detailed overview of the design targets for the hybrid packet buffer architecture. For each target Section 3.2 derives the required architectural features which enable them. Section 3.3 introduces in detail the hybrid memory architecture proposal while Section 3.4 quantitatively evaluates its resource requirements and compares the results to those of related architectures. A dimensioning example for the proposed architecture closes this Chapter.

3.1 Targets

This Section introduces the targets set for the hybrid memory architecture presented in this thesis. The main objective was to **reduce resource requirements** without reducing functionality. The three explicit targets are listed in the following:

- **Small SRAM size** – SRAM is the most critical resource in a hybrid packet buffer. It may even limit feasibility if the required SRAM size is too large. Consequently, the architecture should require a significantly smaller SRAM size than competing architectures. This leads to a better scalability and to reduced costs. The thesis focuses especially on the SRAM size in the architecture's tail part, i. e., the tail buffer size.
- **Minimal DRAM resources** – Due to the extreme bandwidth and capacity requirements DRAM is a critical resource in a packet buffer, too. Therefore, the architecture should fulfill the following requirements:
  - Utilization of the total DRAM capacity, i. e., no suffering from fragmentation
  - Minimization of the total required DRAM bandwidth
  - High DRAM data bus utilization

  The first two are hard requirements to not waste any DRAM resources. The third requirement means, the DRAM should be operated as efficiently as possible to achieve high data bus utilization, as this decreases the total DRAM data bus width. All three requirements lead to a reduced number of required DRAM devices and consequently to **less I/O pins**. This means a better scalability and reduced costs.
- **Deterministic bandwidth** – The architecture should assure deterministic operation and guarantee maximum bandwidth under any traffic condition. This target implies the following properties:
  - No packet loss (packets are accepted as long as there is free buffer space in DRAM)
  - In-order packet delivery
  - Constant packet read latency

  These properties lead to many advantages (cf. Section 2.2.4), e. g., a simple and deterministic packet buffer interface or safety against adversarial attacks.

The *first* target requires a significant reduction in SRAM size compared to other architectures. Accordingly, this will be a unique feature of the presented architecture. The *second* target is elementary to be cost efficient and scalable. According to the discussion in the related work Section other architectures feature the corresponding requirements at most partly. The *third* target is a basic requirement. All architectures presented in the related work Section feature this (the PHSD features this only partly).

This work focuses on the memory architecture for packet buffering. This means, it complies – as competing architectures too – with the packet buffer definition from Section 2.2.1 (page 17) except the fact that they do not provide queuing information to an external scheduler. The following Section introduces the design considerations to meet the targets.

3.2 Design Considerations and Challenges

The targets set require for design consideration to identify necessary functions and properties. Related architectures already fulfill the targets partially (cf. Section 2.5.2). Consequently, we combine their best approaches and enrich them with own ones to form a new architecture that meets all targets simultaneously. In the following for each target the necessary functions and properties are identified.

Deterministic Bandwidth

To achieve a deterministic bandwidth two requirements have to be fulfilled:

Firstly, the utilized MMAs have to guarantee that they process incoming write or read request within bounded time. This implies that used memories (SRAM, DRAM) behave deterministically, i. e., each access takes the same constant duration of time. SRAM always fulfills this. To behave deterministically the DRAM has either to be accessed just once per random access time T or if banking is utilized the banks have to be accessed in a systematic way. This is the task of the MMAs.

Secondly, head and tail buffer (SRAM) of the architecture have to be dimensioned according to the introduced bounded delays of the MMAs. This means, the tail buffer has to be large enough to never drop an incoming packet. Accordingly, the head buffer – which serves as a reorder buffer – has to be large enough so that the system can always deliver requested packets in-order after a constant read latency.

To provide a hard guarantee the upper bounds for both requirements have to be proven, i. e., processing delays as well as tail and head buffer size.

Minimal DRAM Resources

This target intends to minimize the DRAM resources, i. e., capacity, bandwidth, and data bus width. To achieve this, the target architecture has to have the following four properties: no internal fragmentation, no external fragmentation, utilization of banking, and no speedup. Why these properties are necessary and how to achieve them is derived in the following:

No internal fragmentation – Internal fragmentation accounts for most overdimensioning in related architectures. To eliminate it, incoming packet data has to be aggregated to blocks. Then internally, i. e., between tail buffer and DRAM as well as DRAM and head buffer, the required bandwidth is minimal, as always full blocks are exchanged. Analogously for capacity, as DRAM stores only full blocks it requires no overdimensioning. Before leaving the packet buffer the individual packets are reassembled from the blocks. When the packet buffer receives packets already segmented to cells, it has to interpret these and extract the packet data. Concluding, the architecture has to support **aggregation** to eliminate internal fragmentation.

Additionally, aggregation to blocks decouples packet size from DRAM access granularity. This is very desirable, as both can change over generations which could require for large overdimensioning without aggregation, e. g., when the DRAM access granularity doubles or even quadruples in future, internal fragmentation will increase.

No external fragmentation – External fragmentation accounts for the overdimensioning of the DRAM capacity. Overdimensioning is necessary, when the architecture disallows the usage of the total capacity. To eliminate external fragmentation, the architecture and the MMAs have to enable that the data of each maintained queue is spread equally over all utilized DRAMs.

Utilization of banking – The DRAM bus carries blocks from and to DRAM. The more efficiently this is utilized the narrower it can be dimensioned, while providing the same bandwidth. DRAM banking, i. e., interleaved accessing of DRAM banks, allows using the DRAM data bus efficiently, i. e., leads to a high data bus utilization. However, due to complex timing constraints that have to be met the achieved bandwidth highly depends on the sequence of the accessed memory locations. To keep DRAM accesses deterministic the MMAs have to access the individual banks systematically. E. g., when the MMA accesses the banks in a strict round robin manner there is enough time between two accesses to the same bank to prepare the bank for the new access. Usage of banking implies that the architecture has parallel resources – the individual banks.

No speedup – Speedup is the number of times a module works faster then than the line rate. A speedup can be required by the architecture, the MMAs, or just to compensate for inefficient operation. We target to have no speedup. Consequently, having also no internal fragmentation the total required DRAM bandwidth is $2R$. This is the theoretical minimum, as the packet buffer has to store and retrieve packet data with line rate R each. To achieve this, the MMAs and the architecture have to be designed accordingly.

Small SRAM Size

In hybrid memory architectures that guarantee a deterministic bandwidth the required SRAM size is linearly tied to the number of supported queues. Consequently, for larger number of queues the required SRAM size may limit feasibility and increase costs. There is potential to reduce the required SRAM size by exploiting two observations:

- Internal fragmentation accounts for most SRAM overdimensioning – analogously to the DRAM. To eliminate it, incoming packet data has to be aggregated to blocks like already requested before.

- As it will be shown later, the higher the parallelism of an architecture, the smaller is the required block size. Small blocks are earlier ready to be written to DRAM as the time for aggregation is shorter. Consequently, in total fewer blocks can accumulate in the tail buffer. When additionally the DRAM queues dynamically share the tail buffer, the tail buffer size significantly decreases compared to other architectures.

Summary

The most important requirements to meet the targets are listed here:

- MMAs need to operate deterministically
- Incorporate parallelism
- Utilize banking
- Aggregate packets to blocks
- Spread each flow queue equally over all DRAMs
- DRAM queues have to share the tail buffer dynamically (reduces tail buffer size)

Consider that related architectures (cf. Section 2.5.2) also fulfill individual requirements – except the last, which is new. The novelty of the architecture being presented in the next Section is that it fulfills all requirements simultaneously.

3.3 Architecture Proposal

This Section presents the novel hybrid memory architecture fulfilling all targets simultaneously. The first Subsection introduces the overall architecture while the subsequent detail functionality and properties.

3.3.1 Architecture

The proposed architecture is called the Semi-Parallel Hybrid SRAM/DRAM System (SPHSD). The name reflects its two main properties. Firstly, it is a hybrid memory architecture similar to the ones presented in Chapter 2, i. e., it utilizes both, SRAM and DRAM. Secondly, it contains a set of parallel DRAMs but all other building blocks have just a single instance.

The SPHSD is assumed to receive and deliver variable-length packets[1] with a net bandwidth of R each. Utilization of the SPHSD on an input or output line card may require a higher output or input bandwidth, respectively, depending on the speedup of the switch fabric. However, such an imbalance changes only the dimensioning but not the basic architecture or its general properties.

Figure 3.1 shows the functional architecture of the SPHSD. It consists of three main parts: the tail part containing the tail buffer (SRAM), parallel DRAMs, and the head part containing the head buffer (SRAM). Compared to the architectures in the related works Section, this figure also shows the request processing in the head part and therefore obviously looks more complicated. The individual parts of the SPHSD are introduced in the following.

[1]It is common practice in high-speed routers to segment a packet into a stream of constant-size cells on its arrival. On its input the SPHSD reassembles a stream of cells to a packet to perform aggregation. On its output the SPHSD itself delivers a packet as a stream of cells. This means, on the logical level the SPHSD receives and delivers variable-length packets.

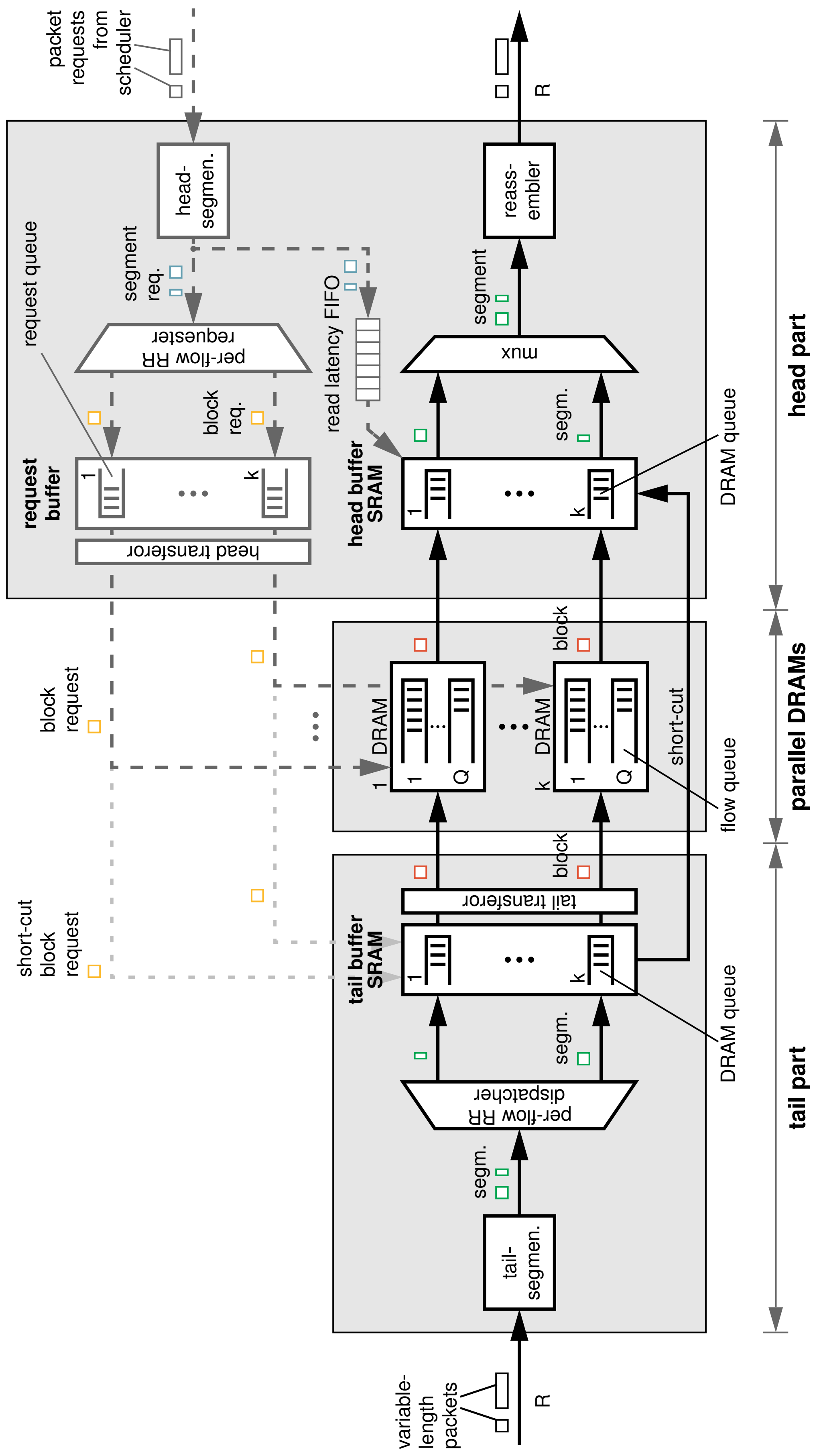


Figure 3.1: Semi-Parallel Hybrid SRAM/DRAM system (SPHSD)

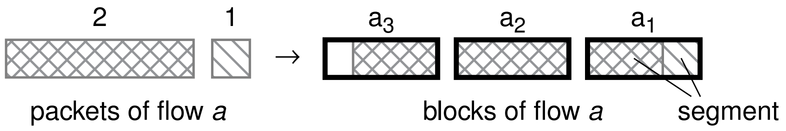


Figure 3.2: Packet segmentation and subsequent aggregation to blocks for one flow

Parallel DRAMs

The core of the SPHSD consists of k parallel DRAMs or DRAM banks, that can be operated independently. A DRAM in Figure 3.1 represents a resource that can be accessed once per random access time T. Consequently, such a DRAM can be also implemented by an individual bank of a DRAM device. This enables in combination with the proper MMA the utilization of banking. For simplicity, the term DRAM is used in the following.

Each DRAM provides $1/k$ of the required bandwidth and contains Q FIFO flow queues, i. e., each logical flow queue is spread over all k DRAMs. The SPHSD aggregates packet data *per-flow* to constant-size blocks. As always full blocks are written to DRAM and read from DRAM the total DRAM bandwidth is dimensioned to $2R$. This is the minimum possible because the SPHSD has to both, accept and deliver packets with line rate R. Hence, each DRAM provides a bandwidth of $2R/k$, i. e., R/k for reading and R/k for writing. The random access time of a DRAM is T and so each DRAM performs one read and one write every $2T$.

Definition 3.1. Block Size *Access time ($2T$) and bandwidth (R/k) of a single DRAM define the block size as*

$$b = \frac{2TR}{k} \tag{3.1}$$

Definition 3.2. Time Slot *A time slot is the time to receive a block of b byte at line rate R.*

$$1 \text{ time slot} = \frac{b}{R} = \frac{2TR}{kR} = \frac{2T}{k}$$

From this it follows that $2T = k$ time slots. In k time slots all DRAMs together can accept k blocks and deliver k blocks simultaneously.

Tail Part

The tail part consists of the building blocks to the left of the DRAMs in Figure 3.1: tail-segmenter, dispatcher, tail buffer, and tail transferor. Its task is to write incoming packet data to the DRAMs in a deterministic way. Therefore it aggregates packet data *per-flow* to constant-size blocks, distributes full blocks to the DRAMs and buffers full blocks when the corresponding DRAM is temporarily overbooked.

In Figure 3.1 from left to right, the **tail-segmenter** receives variable-length packets, divides them into segments and forwards them to the dispatcher. The size of a segment is always chosen in a way that a block gets full – except for the last segment of a packet. In the example in Figure 3.2 the first packet is segmented into one segment of same size, while the second packet is segmented into three segments.

The **dispatcher** distributes segments over k DRAM queues (one for each DRAM). For each flow, each time after one DRAM queue received segments with a total size of b byte, the dispatcher chooses the next DRAM queue. E. g., in Figure 3.2 the first two segments are dispatched to the same DRAM queue so they can be aggregated to block a_1. a_i denotes the i^{th} block of flow a.

The **tail buffer**, which will be implemented with SRAM, maintains the k DRAM queues. A DRAM queue serves two purposes. Firstly, it holds data of non-full blocks during aggregation. Secondly, it holds full blocks in case the corresponding DRAM is temporarily overbooked.

The **tail transferor** knows the state of the DRAM queues. When these contain full blocks the tail transferor transfers the blocks to the corresponding DRAMs.

The MMA in the tail part (tail-MMA) is responsible for the deterministic property of the tail part. Therefore it dictates the behavior of two components: the *dispatcher* and the *tail transferor*. Section 3.3.3 describes the functionality of the tail-MMA in detail.

Head Part

The head part delivers packets requested by an external scheduler in-order with constant read latency. Therefore, for each packet, it requests the blocks from the DRAMs that contain the packet, reorders these when necessary and finally reassembles the packet from the blocks.

The head part consists of the building blocks to the right of the DRAMs in Figure 3.1. These can be grouped into components processing requests (on top right in Figure 3.1) and into components processing packet data (on bottom right in Figure 3.1). Since the SPHSD is symmetric, the tail part and the components processing requests in the head part are similar. The only difference is that the components in the tail part operate on packet data, while the others operate on requests.

In Figure 3.1 beginning from the top right, the external scheduler sends packet requests for variable-length packets. The **head-segmenter** receives these, generates the corresponding segment requests and forwards them to both, requester and read latency FIFO. The packet requests contain the packet's length what makes it simple to generate the segment requests. As each flow queue is accessed in strict FIFO manner the head-requester can generate segment requests that are equivalent to the segments generated by the tail-segmenter before. E. g., the tail-segmenter segments packet 2 of flow a in Figure 3.2 into three segments. Upon receiving the packet request for packet 2 of flow a the head-segmenter will generate the corresponding three segment requests.

The **requester** determines the DRAMs that contain a block with a segment of the packet. If the corresponding blocks were not requested yet due to previously required segments, it generates the block requests for the corresponding DRAMs and forwards them to the request buffer.

The **request buffer** maintains k request queues, i. e., one per DRAM. Its purpose is to hold the block requests in case the corresponding DRAM is overbooked.

The **head transferor** knows the state of the request queues and transfers contained block requests to the corresponding DRAMs. Blocks returned by the DRAMs are written to the corresponding DRAM queues in the head buffer.

The **head buffer** maintains k DRAM queues, i. e., one per DRAM. Thereby it serves two purposes. Firstly, it stores blocks received from the DRAMs until these can be delivered. Storage is necessary because the head buffer serves as a reorder-buffer. Secondly, it holds segments that were not yet requested by the scheduler.

The **read latency FIFO** delays every segment request by a constant time – the read latency of the packet buffer. After this constant time the read latency FIFO requests the individual segments from the head buffer. The head buffer forwards the segments to the **reassembler**, which reassembles the original packets.

The MMA in the head part (head-MMA) is responsible for the deterministic property of the head part, e. g., segments are always available in the head buffer when requested. Therefore it dictates the behavior of two components: the *requester* and the *head transferor*. Section 3.3.4 describes the functionality of the head-MMA in detail.

Switch fabrics usually operate with constant-size cells (cf. Section 2.1.3). Consider the case when the corresponding packet buffer is on an input line card and serves the switch fabric. If now the block size is dimensioned equal to the cell size then the head-segmenter and reassembler can be omitted. Accordingly, the external scheduler then requests blocks (cells) instead of packets.

Shortcut from Tail to Head Part

A side effect of aggregation is that non-full blocks are never written to DRAM. However, for flows with light traffic it may happen that such a non-full block is required in the head part but is not available in the DRAM. A short-cut path from tail to head buffer solves the problem.

The head part decides for each block request if it sends it to the DRAM or to the tail buffer. To make this decision we exploit the fact that flow queues are always read in FIFO manner. Consequently, the head transferor sends the block request to the corresponding DRAM whenever this contains a block of the corresponding queue. Else, when there is no block of the corresponding flow queue in the corresponding DRAM, it sends it to the tail buffer.

3.3.2 Degree of Parallelism

Parameter k defines the number or parallel DRAMs utilized in the architecture – the degree of parallelism. This Section introduces the constraints for choosing k.

The aggregation implemented by the architecture decouples packet size and DRAM access granularity. The tail buffer aggregates incoming packets to blocks and the tail transferor accesses the DRAMs in granularity of one block. The block size $b = 2TR/k$ is inversely proportional to k. Decoupling allows to use k as degree of freedom. The system designer can chose k between a lower and an upper bound, which are derived in the following.

Lower bound of k – The architecture defines the lower bound of $k = 1$, i. e., the system is not parallel any more. For $k = 1$ the block size and the architecture are equal to that of the HSD of Iyer *et al.* [45]. Only the MMAs are still different.

Upper bound of k – The DRAM type utilized defines a meaningful upper bound for k. Each memory has a minimal access granularity, which is the product of its data bus width and the burst length of an access. For example, a DDR3 SDRAM DIMM has 64 data pins and is only operated efficiently with a burst length[2] of 8, i. e., its access granularity is $8 \cdot 64\,\text{bit} = 64\,\text{byte}$. Alternatively, a single DDR3 SDRAM chip has just 4 data pins, i. e., its access granularity is $8 \cdot 4\,\text{bit} = 4\,\text{byte}$.

The block size is an integer multiple of the DRAM access granularity and so is at least equal to it. At a given line rate this determines the upper limit of k, e. g., for $R = 100$ Gbps and $T = 53.1$ ns using DDR3 SDRAM DIMMs $\lceil k \rceil = 21$ and using DDR3 SDRAM chips $\lceil k \rceil = 332$.

The system designer can chose k to optimize resource requirements. Thereby one has to consider the following three main impacts of k on the system:

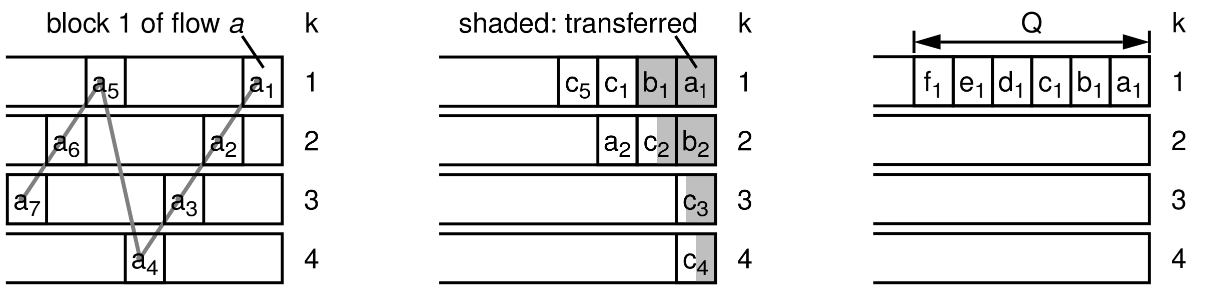
Tail buffer size decreases with increasing k – It will be proven in Section 3.4.1 that when the DRAM queues share the tail buffer dynamically its size decreases with increasing k. Thereby the tail buffer size asymptotically converges towards a lower bound.

Queue management overhead for blocks increases with k – The system operates internally in granularity of blocks. Maintaining queues of these blocks in tail part, head part and DRAMs introduces a management overhead. The block size ($b = 2TR/k$) decreases linearly with increasing k. Consequently, the number of blocks as well as the management overhead increase towards larger k. At some point management overhead outweighs other savings.

k allows minimizing DRAM resources in real implementation – Commercial DRAM chips are only available in a few configurations, e. g., DDR3 SDRAM has always 8 banks and is available with 4, 8 or 16 data pins combined with a few different capacities and bus speeds. The organization of these DRAM chips depends on k. E. g., for $k = 16$ DRAM could be organized in two DRAM groups with 8 banks each, while a DRAM group consist of a set DRAM chips operated in unison. Each organization in combination with an explicit DRAM chip introduces more ore less DRAM overdimensioning. The system designer now can use k to find a DRAM organization that minimizes DRAM overdimensioning. Section 3.5 gives a corresponding detailed example.

Beside these explicit DRAM resources k also allows to influence the pin count to control the DRAMs. The individual DRAM groups are operated independently, i. e., each DRAM group has its dedicated address and control bus. Hence, the higher the number of DRAM

[2]The DDR3 SDRAM standard [29] defines the burst lengths BC4 (4 data words) and BL8 (8 data words). BC4 is a chopped burst and therefore consumes almost the same time but transfers only half as much data words as BL8. Consequently, with BL8 data bus utilization is nearly double as high.



(a) round robin dispatching principle

(b) arrival of blocks from several flows

(c) flows synchronized to round robin scheme

Figure 3.3: Status of the DRAM queues in the tail buffer

groups, the higher the required pin count of the chip implementing the packet buffer. Each DRAM group increases k by at most the number of its banks. Concluding, the lower k, the lower the number of required DRAM groups, the lower the pin count for address and control busses to operate the DRAMs.

3.3.3 Tail Memory Management Algorithm

This Section describes the MMA utilized in the tail part (tail-MMA). This is identical to the tail-MMA proposed by Wang *et al.* in [88], but is utilized in a new architecture. Consequently, it leads to different results. The tail-MMA is implemented by two components: the *per-flow round robin dispatcher* and the *tail transferor*. Dispatcher and transferor work independently. The functionality of these is described in the following.

Per-flow Round Robin Dispatcher

To provide a deterministic bandwidth the SPHSD has to be able to write/read blocks of each flow to/from DRAM with line rate. As only all DRAMs together provide line rate the dispatcher has to distribute the blocks of each flow equally over all DRAMs.

Accordingly, the per-flow round robin dispatcher distributes packet data of each flow block-wise in a round robin manner over all k DRAMs, i. e., every k^{th} block of each flow is dispatched to the same DRAM. As the dispatcher actually dispatches segments, for each flow it chooses the next DRAM if the current already received segments with a total size of b byte. The dispatcher completes writing b byte into the tail buffer in one time slot. To simplify the description we will say in the following, that the dispatcher dispatches blocks. Further, we will call the arrival of b byte for one flow *block arrival*.

Figure 3.3(a) illustrates the consecutive dispatching of blocks of a single flow to the DRAM queues in the tail buffer. The example assumes $k = 4$ DRAMs. Due to the per-flow dispatching each flow queue is distributed over all k DRAMs. Figure 3.4 illustrates this distribution by showing the individual blocks in the flow queues. The example assumes a system with $k = 4$

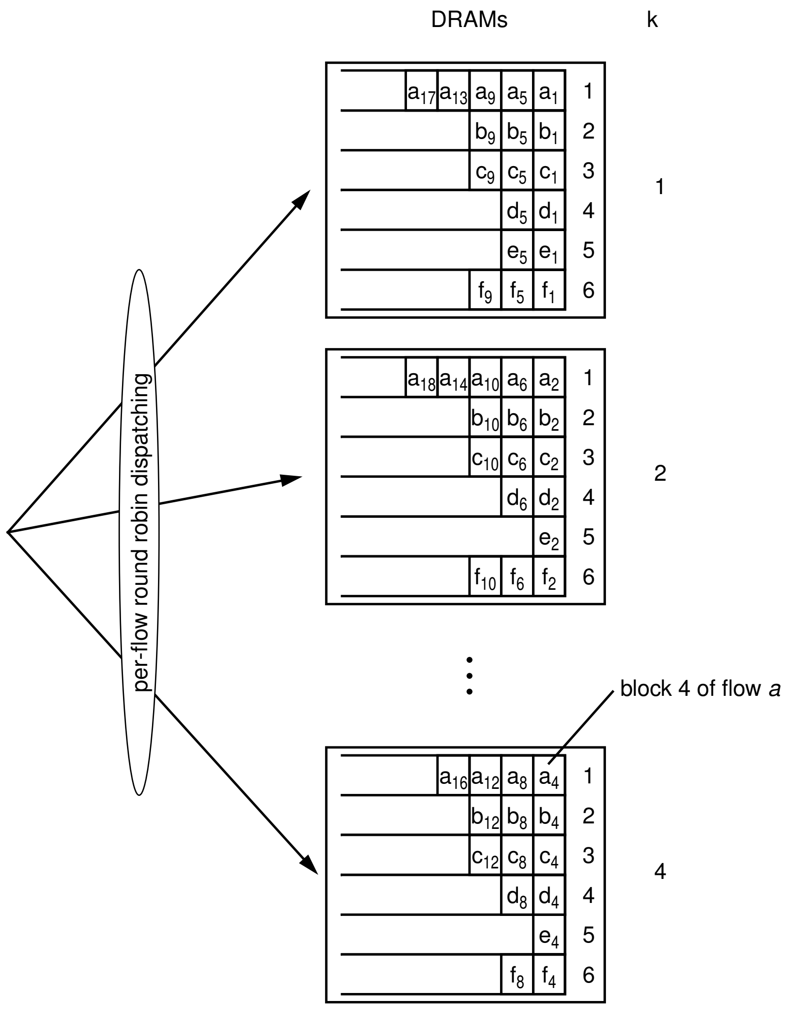


Figure 3.4: Distribution of blocks to the flow queues in the DRAMs; Blocks are dispatched per-flow in a round robin manner to the DRAMs. This example assumes $k = 4$ DRAMs and $Q = 6$ flows named a to f.

DRAMs and $Q = 6$ flows named a to f. Two of the presented architectures in the related work Section (CFDS and PHSD) also use the same per-flow round robin dispatching.

Tail Transferor

The task of the tail transferor is trivial. It transfers full blocks from tail buffer to DRAM as fast as possible in order to keep tail buffer occupation low. Each DRAM completes writing a block in $2T = k$ time slots. The transferor performs parallel writes to all k DRAMs if all DRAM queues have full blocks. With full blocks in each DRAM queue the tail buffer fill level cannot raise, as per time slot, b byte leave and at most b byte arrive.

Example of tail-MMA Operation

The following example illustrates the operation of the tail-MMA. We assume to have $k = 4$ DRAMs, $Q = 6$ flows named a to f and to start from an empty system at the beginning of time slot 1. Figure 3.3(b) shows the tail buffer state after the arrival of a total of 9 blocks from several flows, i. e., the state of the $k = 4$ DRAM queues. To simplify the example we assume that the received packets' size is an integer multiple of the block size. The 9 blocks arrive at the tail buffer back to back in the following order: a_1, b_1, b_2, c_1, c_2, c_3, c_4, c_5, a_2. The dispatcher dispatches block a_1 and b_1 to DRAM queue 1, block b_2 to DRAM queue 2, etc.

In DRAM queues 1 and 2 blocks are accumulated, as due to the round robin dispatching these DRAMs are temporarily overbooked. Simultaneously to the block arrivals, the tail transferor transfers full blocks from the DRAM queues to the corresponding DRAM. The grey shaded area marks blocks that have been transferred to the DRAMs meanwhile. E. g., after block c_4 two more blocks arrive. During these two time slots the transferor transfers block c_4 to DRAM half. It takes $k = 4$ time slots to transfer it completely. The state of DRAM queue 1 explains the following. The first block a_1 arrived in time slot 1 to DRAM queue 1. The snapshot in Figure 3.3(b) shows the state at the end of time slot 9. Accordingly, the transferor could transfer $(9-1)/k = 2$ blocks to DRAM 1, i. e., explicitly blocks a_1 and b_1.

Lemma 3.1. *No more than Q blocks are accumulated per DRAM queue for $P_{min} \approx 0 \ll b$, where P_{min} is the minimal packet size.*

Proof. Assume all Q flows have a nearly full block in the tail buffer. Then each flow receives a minimal size packet completing aggregation of all Q blocks. If all flows are synchronized to the round robin sequence, then all Q blocks are in the same DRAM queue. Figure 3.3(c) shows the case after the sequential completion of the blocks a_1 to f_1.

The per-flow round robin dispatcher assigns every k^{th} block of a flow to the same DRAM queue. However, in k time slots the transferor can remove one block from every DRAM queue and write it to DRAM. So no more than Q blocks are accumulated per DRAM queue. □

3.3.4 Head Memory Management Algorithm

This Section describes the MMA utilized in the head part (head-MMA). Equivalently to the tail, the head-MMA is implemented by two components: the *per-flow round robin requester* and the *head transferor*. Requester and transferor work independently.

The SPHSD architecture is symmetric. Symmetric is the complete tail part to the head part components processing requests (cf. bottom left and top right in Figure 3.1). The main difference between symmetric components is that the ones in the tail part operate on packets, while the ones in the head part operate on packet requests. Accordingly, there is also some symmetry between tail-MMA and head-MMA. The components of the head-MMA are introduced in the following.

Per-flow Round Robin Requester

To be able to retrieve data blocks in-order from the DRAMs the requester implements the equal per-flow round robin scheme as utilized in the tail part.

The requester generates block requests per-flow in a round robin manner for all k DRAMs, i. e., every k^{th} block request for each flow is for the same DRAM. Upon arrival of a segment request it generates a block request for this flow, except when it already generated this block request before. The latter may happen, when a previous segment request already triggered the generation of this block request.

Head Transferor

The head transferor transfers block requests from request buffer to the DRAMs. Each DRAM completes reading a requested block in $2T = k$ time slots. Initiating parallel reads from all k DRAMs the head transferor retrieves from DRAMs k blocks in k time slots. Transferring block requests from a request queue it has two degrees of freedom: when to transfer a block request and which block request to transfer next.

Here two algorithms are introduced. Both process block requests in each request queue in strict FIFO manner. This minimizes the maximal delay that can be introduced to a block request.

1. MiRBD algorithm – The head transferor implements the same algorithm as the tail transferor and is thus also identical to the algorithm proposed by Wang *et al.* in [88]. However, as it is utilized in a new architecture it leads to different results. The head transferor transfers available block requests as soon as possible to the DRAMs to keep request buffer occupation as low as possible. In this thesis the algorithm is called the Minimize Request Buffer Delay (MiRBD) algorithm, as it minimizes the delay introduced by the request buffer. In this case, the properties of the request queues are identical to that of the DRAM queues in the tail part, i. e., Lemma 3.1 holds also here and so no more than Q block requests are accumulated in a request queue. Equivalently to the tail part, due to temporal overbooking of the DRAMs block requests may be processed out of order compared to their generation order. The head buffer reorders the blocks before they leave the packet buffer.

Observation 3.1. *With the MiRBD algorithm's strategy of processing every block request as soon as possible many blocks wait an unnecessary long time in the head buffer. This leads to an unnecessary large required head buffer size.*

$\star$ **Idea.** Keeping block requests as long as possible in the request buffer intuitively leads to less blocks to be reordered. This means, a smaller head buffer is sufficient.

The algorithm proposed in the following bases on this idea.

2. **MaRBD algorithm** – The head transferor behaves inversely to the MiRBD algorithm and delays processing of a block request as long as possible. It is called the Maximize Request Buffer Delay (MaRBD) algorithm. The constraint thereby is to not raise the maximum latency introduced to a block request compared to the MiRBD algorithm, i. e., the system properties will not change. To achieve this, the algorithm has to guarantee that here also **never more the Q block requests accumulate in a request queue**. The algorithm therefore has to suffice two rules. Both define when at the latest the head transferor has to process the headmost block request in a request queue. The rules are orthogonal as they monitor different properties. The head transferor has to start transferring the headmost block request in a request queue as soon as one of the rules triggers this.

   - **Rule 1: Maximally delay a block request** – The queue state defines how long processing of the headmost block request can be delayed. For example, assume that a request queue contains just one block request and no other request arrives to this queue in the next Qk time slots. Then the head transferor can delay this by $(Q-1)k$ time slots. When the queue contains Q requests, processing cannot be delayed at all. In both border cases the maximum latency introduced to a block request is Qk time slots.

   - **Rule 2: Consider round robin requesting state to never accumulate more than Q blocks** – Processing of the headmost block request in a request queue depends on the current round robin requesting state of the individual flows. The latter contains for each flow the information which DRAM received the last block request and how much of the corresponding block data was requested up to now. The transferor has to keep the request queue fill level such low, that independent of the outgoing traffic pattern newly generated block requests cannot raise the fill level of a request queue above Q. This is simple due to the deterministic behavior of the per-flow round robin requesting.

     For example, assume that the last block request of all queues was enqueued in request queue i. Up to now only a minimal size packet P_{min} was requested from each of the corresponding data blocks. Before now the packet requests from the external scheduler can trigger the generation of Q block requests for request queue $i+1$ it has to request all the remaining data from the previous Q blocks, i. e., $Q \cdot (b - P_{min}) \approx Qb$. Requesting Qb data takes Q time slots. In these Q time slots the head transferor can remove Q/k block requests from request queue $i+1$. Consequently, at this point in time request queue $i+1$ is allowed to contain up to Q/k block requests. Analogously, until request queue i can receive Q new block requests it takes at least Qk time slots. Consequently, at this point in time request queue i is allowed to contain up to $Qk/k = Q$ block request.

A head-MMA that utilizes the MiRBD head transferor algorithm will be called *MiRBD head-MMA* in the following. Accordingly, a head-MMA that utilizes the MaRBD head transferor algorithm will be called *MaRBD head-MMA*. The required head buffer sizes will be evaluated in Section 3.4.2. As requests are negligible in size compared to blocks, the request buffer is not considered further.

3.4 Quantitative Assessments

As introduced in Chapter 2 the main metrics of hybrid memory architectures for packet buffers are the required tail and head buffer size as well as the introduced read latency. For a quantitative assessment in the following the upper bounds for these will be derived and proven. Further, these metrics are compared to those of related architectures. Finally, DRAM resource requirements are also compared to those of related architectures.

For the following proofs we assume that the minimal packet size P_{min}, that can arrive to or depart from the packet buffer, is approximately 0. This is a worst-case approximation that slightly raises the bounds but simplifies the proofs.

3.4.1 Tail Buffer Size

The proposed hybrid memory architecture utilizes a tail buffer to hold blocks in times when DRAMs are overbooked. This Section first derives an upper bound for the tail buffer size and then assesses it quantitatively.

3.4.1.1 Upper Bound

Observation 3.2. *As soon as a DRAM queue contains at least one full block the transferor starts transferring it to the corresponding DRAM. Consequently, not all DRAM queues can be full simultaneously, i. e., contain Q blocks.*

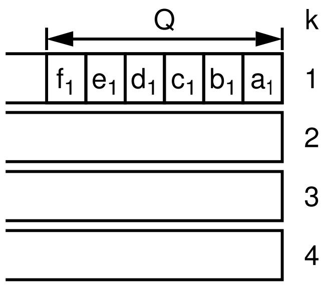
⋆ **Idea.** We can let the DRAM queues dynamically share the tail buffer and thereby significantly decrease the total required tail buffer size.

Theorem 3.1. *If the DRAM queues dynamically share the tail buffer by utilizing dynamic memory allocation, the upper bound for the tail buffer size in blocks is*

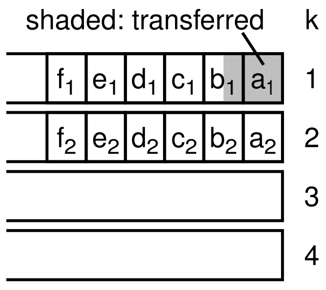
$$\boxed{Q\frac{(k+1)}{2} \qquad (3.2)}$$

Proof. We assume that packets arrive at the packet buffer continuously with full line rate R. This represents the worst case if we want to show that the buffer size is bounded. The proof consists of four steps leading to Lemma 3.2, 3.3, 3.4 and 3.5.

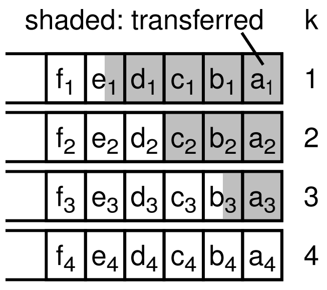
Observation 3.3. *As long as any DRAM is idle because its DRAM queue contains no full blocks, tail buffer size will grow.*



(a) $t = Q$ time slots



(b) $t = 2Q$ time slots



(c) $t = kQ$ time slots

Figure 3.5: Tail buffer status during the arrival of traffic pattern P1

The worst-case traffic pattern maximizes DRAM idle time and by this it defines the upper bound for the tail buffer size. In the following we define a traffic pattern and prove that it is the worst-case traffic pattern, as it maximizes required buffer size.

Definition 3.3. Traffic Pattern P1 *The traffic pattern P1 has the following properties:*

- *Q blocks accumulate to one DRAM queue according to Lemma 3.1.*
- *This consecutively happens $\geq k$ times*

Figure 3.5 gives an example for P1 assuming $Q = 6$ and $k = 4$. Starting from an empty tail buffer at $t = 0$ after $Q - \varepsilon$ time slots no block is fully aggregated yet. Until now all DRAMs are idle. At $t = Q$ time slots all Q blocks get full in the first DRAM queue (cf. Figure 3.5(a)).

At $t = 2Q$ time slots Q blocks get full in DRAM queue 2 (cf. Figure 3.5(b)). Up to now DRAM queue 1 transferred $Q/k = 1.5$ blocks to DRAM 1. Transferred blocks are shaded in the figure, i. e., these are not any more in the tail buffer. Up to now all other DRAMs are still idle. Figure 3.5(c) shows the tail buffer status at $t = kQ$ time slots.

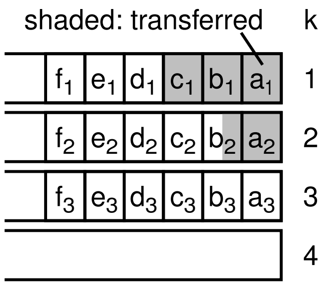
Lemma 3.2. *With P1, starting from an empty tail buffer the maximally required tail buffer size in blocks is*

$$Q \cdot \frac{(k+1)}{2} \tag{3.3}$$

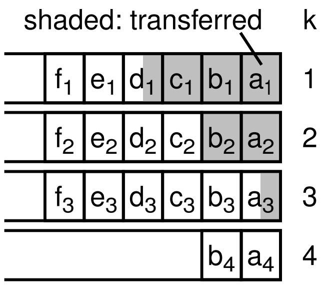
Proof. The non-shaded area in the Figure 3.5(c) represents the required tail buffer size. Based on this we can calculate the buffer size for arbitrary k and Q.

Accumulation of Q blocks in each of the DRAM queues takes Q time slots. The transferor removes in Q time slots Q/k blocks from any DRAM queue with full blocks. In this example, at $t = kQ$ time slots DRAM queue i has $Q\frac{i}{k}$ full blocks, e. g., DRAM queue $k-1$ has $Q\frac{k-1}{k}$ full blocks. Summing up the blocks of all DRAM queues gives us

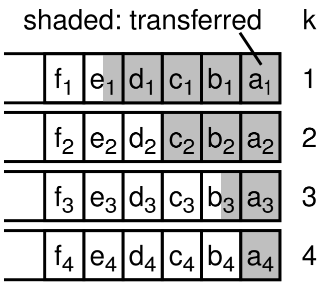
$$S_{P1} = Q \cdot \sum_{i=1}^{k} \frac{i}{k} = Q \cdot \frac{(k+1)}{2}$$



(a) $t = 3Q$ time slots



(b) $t = 3Q+2$ time slots



(c) $t = kQ$ time slots

Figure 3.6: Tail buffer status during the arrival of traffic pattern P2

At any time $t = xQ$ time slots, with $x > k$ the buffer size is equal to S_{P1} because the fill levels just rotate through the DRAM queues. E. g., starting from Figure 3.5(c), at $t = (k+1)Q$ time slots DRAM queue 1 will have the fill level of DRAM queue k, 2 that of 1 and so on. As all DRAM queues have full blocks all the time, the tail transferor can remove blocks with full line rate. This leaves enough free buffer space for new segments arriving with line rate. □

Now we show that starting from an empty tail buffer any traffic pattern different from P1 leads to a lower bound for the tail buffer size.

Definition 3.4. Traffic Patter P2 *The traffic pattern P2 includes all possible traffic patterns except P1.*

Lemma 3.3. *With P2, starting from an empty tail buffer the maximal required tail buffer size is always less than with P1.*

Proof. Starting from an empty tail buffer, with P2 some blocks get full earlier than with P1. Consequently, the transferor also starts writing blocks to DRAM earlier. This always leads to a smaller maximal required tail buffer size than P1.

Figure 3.6 gives an example for P2. We assume $Q = 6$ and $k = 4$. Starting from an empty tail buffer, up to $t = 3Q$ time slots in this example, there is no difference to P1 (cf. Figure 3.6(a)). At $t = 3Q+2$ time slots already 2 blocks get full in DRAM queue 4 (cf. Figure 3.6(b)). At $t = kQ$ time slots 4 further blocks get full in DRAM queue 4 (cf. Figure 3.6(c)). The difference to P1 is, that the tail transferor could start writing blocks to DRAM 4 earlier. This leads to a smaller total tail buffer size compared to P1. Figure 3.6(c) visualizes this by having more shade area than Figure 3.5(c). □

Now we consider starting from a *non-empty tail buffer*.

Lemma 3.4. *Starting from any valid non-empty tail buffer status P1 does not raise the tail buffer size from Lemma 3.2.*

Proof. The proof requires two definitions.

Definition 3.5. System A *System A, is a system that receives from the beginning only P1. F_{Ai} denotes the length of DRAM queue i in its tail buffer. Figure 3.5 depicts the tail buffer status of such a system.*

Considering System A, independent how long P1 is received, the required buffer size is bounded (cf. Lemma 3.2).

Definition 3.6. System B *System B, is a system that receives from the beginning only P2. Then packet arrivals of the flows synchronize to the round robin scheme so that the system can receive P1 in the following. F_{Bi} denotes the length of DRAM queue i in its tail buffer. Figure 3.6 depicts the tail buffer status of such a system.*

Now we compare the tail buffer status of systems A and B. We consider the status of system A depicted in Figure 3.5(c) and the status of system B depicted in Figure 3.6(c). System B is at the point in time before it starts receiving P1. Comparison shows that $F_{Ai} \geq F_{Bi}, \forall i \in \{1,2,\ldots,k\}$. Concluding, when system B starts now receiving P1 then this cannot raise the upper bound from Lemma 3.2 as it has a better starting position than system A. □

Lemma 3.5. *Starting from any valid non-empty tail buffer status P2 does not raise the tail buffer size from Lemma 3.2.*

Proof. We have to distinguish if P1 or P2 introduces the starting non-empty tail buffer status. For the case that P2 introduces it, Lemma 3.3 already proves that P2 cannot raise the bound from Lemma 3.2.

The case that P1 introduces the non-empty tail buffer status is considered in the following. We make the following two observations concerning the increase and decrease of a system's tail buffer size.

Observation 3.4. *Receiving packets at full line rate R, the tail buffer size of a system* ***cannot decrease*** *as the DRAM bandwidth available for writing is also R. This is true independent of the incoming traffic pattern.*

Observation 3.5. *Receiving packets at full line rate R, the tail buffer size of a system* ***cannot increase*** *as long as each DRAM queue has one ore more full blocks. This is true independent of the incoming traffic pattern because the transferor can write blocks to DRAMs with line rate R.*

We show now, that in the considered scenario at full line rate none of the DRAM queues can run empty of full blocks. According to the last observation the tail buffer size then also cannot increase.

The tail buffer status in Figure 3.5(c) is the starting point, i. e., a status introduced by P1. To see if a DRAM queue i can run empty of full blocks we need a traffic pattern that maximizes the waiting time of DRAM queue i for full blocks. Only P1 has this property. E. g., the maximum time for DRAM queue 1 until a new block gets full is Q time slots. The Q/k blocks in DRAM queue 1 are exactly enough not running empty in these Q time slots. From this we conclude, that here with P1 no DRAM queue can run empty. Consequently, with P2 also no DRAM queue can run empty, because with P2 blocks get full earlier compared to P1. Concluding, P2 does not raise the tail buffer size from Lemma 3.2. □

From Lemma 3.2, 3.3, 3.4 and 3.5 it follows, that independent of the received traffic pattern (i. e., only P1, only P2, first P1 then P2, or first P2 then P1) the upper bound of the required tail buffer size is that given in Lemma 3.2. According to the definitions of P1 and P2 these cover together all existing traffic patterns. Finally, P1 leads to the maximum tail buffer size from Lemma 3.2 and is therefore the worst-case traffic pattern. □

For sake of completeness, the tail buffer size is also derived for the case, where the DRAM queues *do not share* memory.

Theorem 3.2. *If the tail buffer is statically divided in k partitions (one for each DRAM queue) and each partition utilizes dynamic memory allocation, then the upper bound for the tail buffer size in blocks is*

$$\boxed{Qk \qquad (3.4)}$$

Proof. We know from Lemma 3.1 that no more than Q blocks can accumulate per DRAM queue. With k DRAM queues the upper bound is Qk blocks.

This already includes the memory required for aggregation. Assume that at $t_0 = Q$ time slots Q blocks got full in a DRAM queue, i. e., the DRAM queue contains a total of Q full blocks (cf. Figure 3.5(a)). Beginning from t_0 the transferor removes data from that DRAM queue, i. e., one block (b byte) per k time slots. This DRAM queue will receive any further segment from any flow earliest during the time slot $t_0 + k$. At the end of time slot $t_0 + k$ new b byte may be received but the transferor also removed b byte. Concluding, the tail buffer size is sufficient.

Finally, requirement for dynamic memory allocation is proven. Assume, the same example as before, i. e., a tail buffer partition is completely full by storing Q blocks. The tail transferor removes the blocks one by one from the tail buffer. However, new packet data for this DRAM queue may arrive **for any flow** in any granularity, so aggregation of several new blocks may be started. Incoming packet data of the different flows have to share the freed buffer space. This is only possible by utilizing dynamic memory allocation. □

3.4.1.2 Assessment

This Section assesses the required tail buffer size of the SPHSD. The previous Section introduced upper bounds for two different organizations of the tail buffer. Here only the first organization is evaluated (cf. Theorem 3.1 on page 80) as only this reduces tail buffers size significantly compared to other architectures.

The following paragraphs first evaluate the dependency of the tail buffer size on the parameter k. Then it compares the tail buffer size of the SPHSD to that of other architectures.

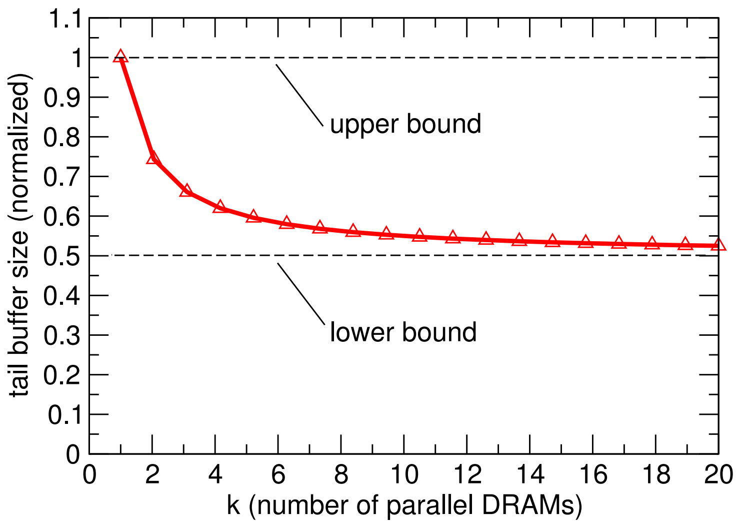


Figure 3.7: Tail buffer size of SPHSD as a function of parameter k

Evaluation of the Dependency on k

According to Theorem 3.1 the tail buffer size of the SPHSD **in blocks** is

$$S_{tail-SPHSD} = Q\frac{(k+1)}{2}$$

while a block has the size **in byte**

$$b = \frac{2TR}{k} = \frac{B}{k} \tag{3.5}$$

where $B = 2TR$ (cf. page 51). Expressed **in byte** the tail buffer size is

$$S_{tail-SPHSD} = QB\frac{(k+1)}{2k} \tag{3.6}$$

Figure 3.7 shows how the tail buffer size depends on k – the number of parallel DRAMs used in the system. To remove dependency from explicit system parameters like T and R the tail buffer size is printed normalized, i. e., for $k = 1$ the tail buffer size is 1.

$S_{tail-SPHSD}$ asymptotically converges with increasing k towards the lower bound of

$$S_{tail-SPHSD-min} = QB\frac{1}{2} \tag{3.7}$$

This means, the tail buffer size decreases by 50 %. Before discussing meaningful choices, recall that the system designer can freely choose k between a lower and an upper bound (cf. Section 3.3.2). According to Figure 3.7 $S_{tail-SPHSD}$ comes close to its lower bound with already small values for k. For example, with $k = 10$ $S_{tail-SPHSD}$ already decreases by 45 %. Concluding, the queue management overhead that increases with k (cf. Section 3.3.2) is acceptable as reasonable k are small.

Comparison to Other Architectures

In the following the tail buffer size of the SPHSD is compared to the tail buffer sizes of the architectures introduced in the related work Section: HSD, PHSD, and CFDS. For a simpler comparison we use as far as possible formulas that are cleared from effects that come from the author's different assumptions on arriving data granularity (cf. Table 2.5 in Section 2.5.2.4). Further, we disregard the *65 byte problem* to which PHSD and CFDS are suffering as this requires assumptions about the minimal packet size and the cell size used in these systems. Regarding it would mean for these systems a resource increase of up to a factor of two, depending on the implementation.

For comparison we assume a SPHSD with $k = 16$. According to the previous paragraphs this is a reasonable value. This leads to a tail buffer size of

$$S_{tail-SPHSD-k16} = QB\frac{16+1}{2\cdot 16} = 0.53\,QB$$

The HSD has a tail buffer size in byte of $S_{tail-HSD} = QB$. The minimal tail buffer size of the PHSD in byte is $S_{tail-PHSD} = Qbk = QB$. Consider that the parameter k of the PHSD also defines the parallelism of the system but it influences the system properties differently.

With $k = 1$ in the SPHSD (Eq. (3.6)) the tail buffer size is equal to HSD and PHSD:

$$S_{tail-SPHSD-k1} = S_{tail-HSD} = S_{tail-PHSD}$$

However, a SPHSD with $k = 16$ requires a 47 % smaller tail buffer size compared to HSD and PHSD.

Garcia *et al.* analyze only the head buffer size of the CFDS. However, according to our understanding of the CFDS its tail and head buffer size should be roughly similar. Consequently, to allow a rough comparison to the CFDS we assume $S_{tail-CFDS} = S_{head-CFDS-ECQF}$.

Comparison with CFDS requires an assumption about the number of individual banks available in the utilized DRAM. Currently relevant DRAMs for packet buffers are DDR3-SDRAM and RLDRAM II (cf. Section 2.3.4). Both feature $M = 8$ banks (cf. Table 2.2 in Section 2.3.3). Future DRAMs tend to have more banks, e. g., RLDRAM III is announced to have 16 banks.

Here we consider $M = 8$, $M = 16$, and $M = 32$ banks to cover current DRAMs and ones being possibly available in closer and farer future. The corresponding tail buffer sizes in the CFDS are $0.75\,QB$, $0.62\,QB$, and $0.43\,QB$, respectively. Comparison shows that for $M = 8$ the SPHSD requires a 29 % and for $M = 16$ a 14.5 % smaller tail buffer size. For $M = 32$ the CFDS requires a smaller tail buffer size than the SPHSD. However, such forecasts are difficult as corresponding DRAMs in farer future may not only have a larger number of banks, but also other properties, compared to today's DRAMs.

Only the CFDS suffers from external fragmentation in the DRAM. To make comparison more fair we consider also a CFDS that utilizes its queue renaming mechanism, which alleviates external fragmentation (cf. Section 2.5.2.2, page 53). This CFDS requires $P > Q$ flow queues to be managed, while P replaces Q in the formulas. We assume $P = 1.5Q$. For $M = 8$, $M = 16$, and

$M = 32$ banks this leads to a tail buffer size in the CFDS of $0.75\,PB = 0.75 \cdot 1.5QB = 1.125\,QB$, $0.93\,QB$, and $0.645\,QB$, respectively. Comparison shows that the SPHSD requires a 53 %, 43 %, and a 17.8 % smaller tail buffer size, respectively.

Summarizing, the tail buffer size of the SPHSD decreases with increasing k. Thereby it converges towards a lower bound. Reasonable values for k are in the range of 8 to 20 as these decrease the tail buffer size close to the lower bound and at the same time keep queue management overhead acceptable.

The SPHDS significantly outperforms other architectures with respect to the tail buffer size. Quantitatively, the SPHSD reduces the tail buffer size by 47 % to 53 % depending on the compared architecture. This result disregards the 65 byte problem to which PHSD and CFDS suffer. Regarding it would further raise reduction compared to these systems by up to a factor of two.

3.4.2 Head Buffer Size

The SPHSD utilizes a head buffer to enable a deterministic output behavior of the system. This Section first derives the required head buffer size and then quantitatively assesses it.

3.4.2.1 Upper Bound

Section 3.3.4 introduced two different algorithms for the head transferor: the MiRBD algorithm and the MaRBD algorithm. These lead to different head buffer sizes and are considered individually.

MiRBD Head Transferor Algorithm

Theorem 3.3. *If the head buffer utilizes dynamic memory allocation and the architecture utilizes the MiRBD head transferor algorithm, then the upper bound for the head buffer size in blocks is*

$$\boxed{Q(k+1) \qquad (3.8)}$$

Proof. As described in Section 3.3.1 the head buffer stores data for two purposes. Firstly (a), it stores blocks received from the DRAMs until these can be delivered to provide a constant read latency. Storage is necessary because the head buffer serves as a reorder-buffer. Secondly (b), it stores segments that were not yet requested by the scheduler. The required memory sizes for (a) and (b) are derived individually.

Memory size for (a) depends on the maximal delay a block request can experience in the request buffer plus the time for its retrieval from DRAM.

Lemma 3.6. *The maximal delay a block request can experience in the request buffer plus the time for retrieval of the corresponding block from DRAM is in time slots*

$$\boxed{Qk \qquad (3.9)}$$

Proof. In a DRAM queue in the tail part up to Q blocks are accumulated (cf. Lemma 3.1). Due to symmetry, in a request queue also up to Q block requests are accumulated. The head transferor can read from one DRAM one block every k time slots. Concluding, the maximum delay is Qk time slots, i. e., $(Q-1)k$ time slots queuing delay in the request queue and k time slots for retrieving the Q^{th} block from DRAM. □

From Lemma 3.6 we derive the following to support reordering and a constant read latency at the output interface: *every pair of block request and corresponding block has to be delayed in total Qk time slots*. This delay may be completely introduced by the request buffer and retrieving the block from DRAM (cf. Lemma 3.6). However, under specific traffic patterns the request buffer introduces no queuing delay because no block requests accumulate. In this case the head buffer is responsible to introduce the complete queuing delay of $(Q-1)k$ time slots.

A corresponding traffic pattern is easy to find. E. g., starting from an empty request buffer the scheduler requests packets of a single flow with a total size of at least Qk blocks. Figure 3.8 illustrates this case and shows the status of request buffer and head buffer at the end of time slot Qk. The example assumes $k = 4$ DRAMs and $Q = 6$ flows. Under this request pattern block requests are already generated at the beginning of a time slot. Accordingly, block request a_{Qk} (here a_{24}) arrives to the request buffer at the beginning of time slot Qk. Until the end of this time slot the head transferor processes this block request by $\frac{1}{k}$ (here $\frac{1}{4}$).

According to Figure 3.8 the head buffer stores $(Q-1)k$ completely received blocks. This is required to introduce the necessary queuing delay. The k partly received blocks in the head buffer are required to match the bandwidth gap between a single DRAM and the output line rate. Summed and rounded up, the head buffer requires for purpose (a) maximal a size of Qk blocks.

Memory size for purpose (b) is maximized, when in the given example the $Q-1$ other flows each have one segment of nearly the size of a full block available in the head buffer, i. e., nearly $Q-1$ blocks. Summed and rounded up, the upper bound for the head buffer size is $Q(k+1)$ blocks. □

MaRBD Head Transferor Algorithm

As already mentioned the MaRBD head transferor algorithm has the potential to reduce the required head buffer size. We have shown experimentally by a software implementation that when utilizing the MaRBD algorithm *and* the DRAM queues share the head buffer dynamically,

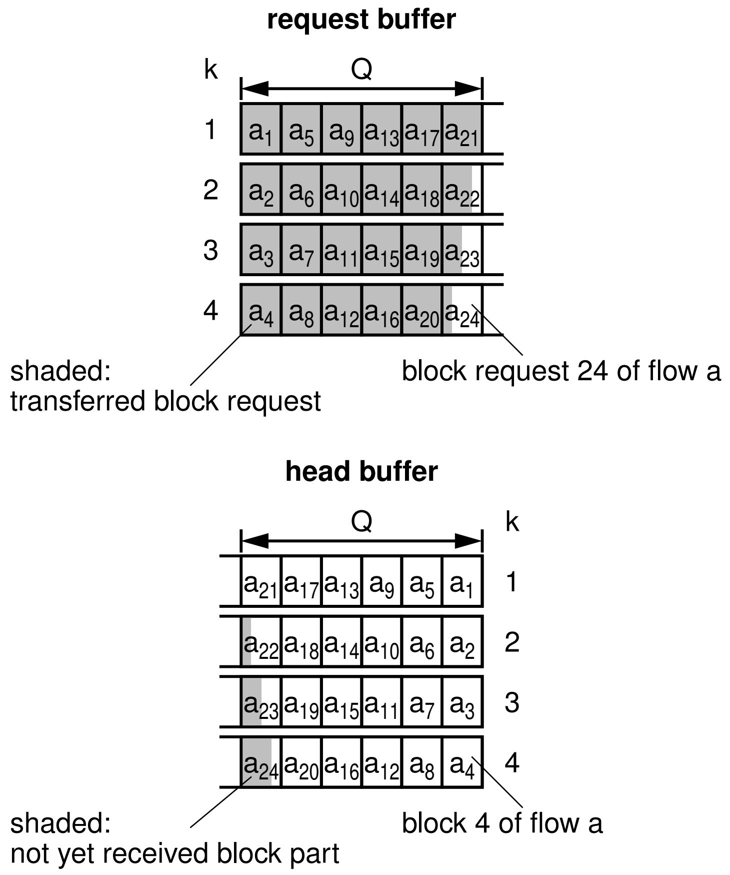


Figure 3.8: Status of request buffer and head buffer: case with maximal head buffer size; Arriving traffic pattern: consecutive request of Qk blocks of a single flow; Figure shows state after t = Qk time slots. The example assumes $k = 4$ DRAMs and $Q = 6$ flows.

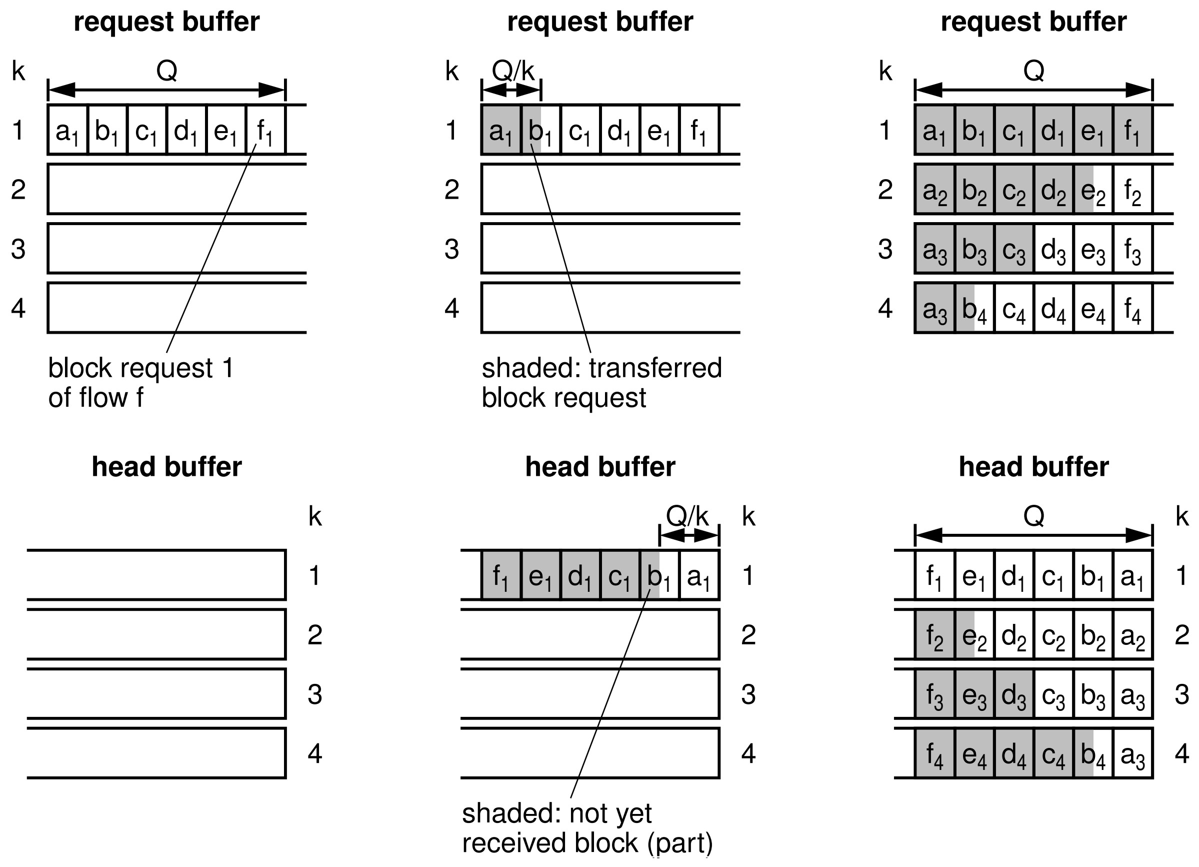
the head buffer size decreases by roughly 50 % compared to Theorem 3.3 for reasonable k, i. e., to $\approx Q\frac{(k+1)}{2}$.

As this thesis focuses on the tail part no further investigations of the exact upper bound for the head buffer size utilizing the MaRBD algorithm are necessary. For a real implementation this has to be caught up to be able to provide a bandwidth guarantee.

The following paragraph sketches in two steps the idea how this saving is achieved. First the theoretical lower bound of the head buffer size's upper bound is derived. Then it is shown how the MaRBD algorithm preserves this.

Lemma 3.7. *The theoretical lower bound of the head buffer size's upper bound when using per-flow round robin dispatching is in blocks*

$$\boxed{Q\frac{(k+1)}{2} \qquad (3.10)}$$



(a) $t = \varepsilon$ time slots (b) $t = Q$ time slots (c) $t = kQ$ time slots

Figure 3.9: Request buffer and head buffer status during the arrival of requests according to traffic pattern P1; The example assumes $k = 4$ DRAMs and $Q = 6$ flows.

Proof. The lower bound of the head buffer size's upper bound is introduced by a traffic pattern where the MaRBD algorithm is without effect, i. e., no block request is delayed additionally. This means under this traffic pattern the MaRBD algorithm behaves identically to the MiRBD algorithm.

From symmetry we know that the traffic pattern P1 (cf. Definition 3.3) that maximizes tail buffer size also maximizes request buffer size. A maximal request buffer size leads to maximized queuing delays for the block requests. According to the proof of Theorem 3.3 on page 87 request buffer and head buffer introduce together a delay of $(Q-1)k$ time slots to every pair of block request and block. Concluding, when the request buffer size is maximal the head buffer size is minimal.

Starting from an empty system Figure 3.9 shows P1 arriving to the head part. During the short time period ε, with $\varepsilon \approx 0$, the external scheduler generates Q minimal size packet requests. The requester accordingly generates the Q block requests which all accumulate in request queue 1. Figure 3.9(a) shows request buffer and head buffer state at $t = \varepsilon$ time slots. During the next Q time slots the head transferor can process Q/k of the block request from request queue 1. Accordingly, DRAM queue 1 in the head buffer holds the Q/k blocks retrieved from DRAM. Figure 3.9(b) shows the state at $t = Q$ time slots. Figure 3.9(c) shows the state at $t = Qk$ time slots, i. e., just before data starts leaving the head buffer. The non-shaded area marks the minimal head buffer size. This is in blocks $Q\frac{(k+1)}{2}$. Beyond $t = Qk$ time slots the head buffer size is constant as data arrives and leaves with line rate. □

According to the proof of Theorem 3.3 request buffer and head buffer introduce together a delay of $(Q-1)k$ time slots to every pair of block request and block. The MaRBD algorithm allows now to introduce roughly half of this delay in the request buffer by processing the block requests delayed. Consequently, for reasonable k the head buffer has to hold roughly only half of the blocks reducing its size to $\approx 50\,\%$ compared to Theorem 3.3.

To validate this result, the author implemented a software model of the head part. The model operates in time slots and in granularity of blocks, i. e., each time slot one block request may arrive and one block may be delivered simultaneously. To account for the memory required due to de-aggregation – what is not covered by the model – one has to add $Q-1$ blocks to the head buffer size delivered by simulation model (cf. proof of Theorem 3.3 on page 87). The author used the following request patterns for validation: only requests of one flow (worst-case traffic pattern for the MiRBD head transferor algorithm), P1 (cf. Definition 3.3 on page 81), and random patterns. All simulations acknowledged the result.

The MaRBD algorithm derives its decision to forward the headmost block request to the DRAMs or not on a simple arithmetic. This bases on the arrival time of the headmost block and the fill levels of the request queues. Concluding, high performance implementations of the MaRBD algorithm should be easily possible in hardware.

3.4.2.2 Assessment

This Section assesses the required head buffer size of the SPHSD. Thereby it addresses both head transferor algorithms as they lead to different head buffer sizes. For each it first evaluates dependency on the parameter k and then shows comparison to other architectures.

MiRBD Head Transferor Algorithm

According to Theorem 3.3 the head buffer size of the SPHSD using the MiRBD head transferor algorithm is **in blocks**

$$S_{head-SPHSD-MiRBD} = Q(k+1)$$

while Eq. (3.5) gives the size of a block. Expressed **in byte** the head buffer size is

$$S_{head-SPHSD-MiRBD} = QB(1+\frac{1}{k}) \tag{3.11}$$

Depending on k $S_{head-SPHSD-MiRBD}$ varies between the lower bound QB for $k \to \infty$ and the upper bound $2QB$ for $k = 1$. With reasonable k like derived during assessment of the tail buffer size (cf. Section 3.4.1.2) the head buffer size comes close to its lower bound, e. g., for $k = 16$ $S_{head-SPHSD-MiRBD-k16} = 1.0625\,QB$.

In the following the head buffer size of the SPHSD is compared to the head buffer sizes of HSD and CFDS. For a simpler comparison we use as far as possible formulas that are cleared from effects that come from the author's different assumptions on arriving data granularity (cf. Table 2.5 in Section 2.5.2.4). Further, we disregard here the *65 byte problem* to which CFDS suffers due to the same reason as in Section 3.4.1.2.

The HSD has a head buffer size of $S_{head-HSD-ECQF} = QB$. Assuming DRAMs with $M = 8$ banks the head buffer size of the CFDS is in byte $S_{head-CFDS-ECQF} = 0.75\,QB$. On the one hand, HSD marginally and CFDS somewhat more outperform the SPHSD regarding the head buffer size. On the other hand, the MMAs utilized in HSD and CFDS require a direct write path from the packet buffer input to the head buffer. This increases the required bandwidth to their head buffer significantly compared to SPHSD (cf. Section 2.5.2.1 and Section 2.5.2.2), i. e., HSD and CFDS require $3R$ while the SPHSD requires $2R$.

Equivalently to the assessment of the tail buffer size, to make comparison fairer, here also additionally a CFDS is considered that suffers from less external DRAM fragmentation. We assume $P = 1.5Q$. Assuming DRAMs with $M = 8$ banks the head buffer size of this CFDS is in byte $S_{head-CFDS-ECQF} = 1.125\,QB$. Now the SPHSD slightly outperforms the CFDS.

The PHSD has no head buffer, i. e., it is not comparable.

MaRBD Head Transferor Algorithm

Utilizing the MaRBD algorithm the head buffer size is roughly similar to the tail buffer size including the dependence on parameter k. Related architectures (HSD and CFDS) have also similar head and tail buffer sizes. Concluding, the SPHSD reduces head buffer size roughly by the same amount as the tail buffer size, i. e., up to 50 %.

Summarizing, the MiRBD head transferor algorithm leads – regarding the head buffer size – to a marginal decrease of 12 % compared to CFDS and a marginal increase of 6 % compared to HSD. The MaRBD head transferor algorithm seems to reduce the head buffer up to 50 % compared to other architectures. As this thesis focuses on the tail part, the latter is not proven. Independent of its size the head buffer in the SPHSD requires a lower bandwidth compared to HSD and CFDS, i. e., 2R instead of 3R.

3.4.3 Read Latency

The SPHSD provides a constant read latency at its output interface. As already defined, the read latency is the time between receiving a read request from an external scheduler and delivering the corresponding packet. The maximal read latency that the architecture introduces to any request defines its value. This Section first derives the constant read latency of the architecture and then quantitatively assesses it.

3.4.3.1 Derivation

Theorem 3.4. *The SPHSD introduces a constant read latency in time slots of*

$$\boxed{Qk \qquad (3.12)}$$

Proof. The total read latency is the sum of the individual maximum latencies introduced by head part and tail part. These are derived in the following. Thereby we assume that all components operate ideally, i. e., introduce no delay for processing. Consequently, we consider only queuing delays and the time for accessing the DRAMs. Further we assume that a DRAM access and short-cutting a block request from tail to head buffer introduces the same latency.

Lemma 3.8. *The tail part introduces no latency.*

Proof. The short-cut path can be used to transfer any block from tail to head buffer, i. e., full and non-full blocks. As the tail part immediately delivers every block requested via short-cut, the tail part introduces no latency. □

Lemma 3.9. *The head part introduces a constant read latency in time slots of*

$$\boxed{Qk \qquad (3.13)}$$

Proof. The maximal delay a block request can experience in the request buffer plus the time for retrieval of the corresponding block from DRAM is Qk time slots (Lemma 3.6). A corresponding block experiences no delay for reordering in the head buffer, as this is already the block with the largest latency. To provide a constant read latency the head buffer delays blocks of which the corresponding block requests were delayed less than maximal in the request buffer. Concluding, the constant read latency introduced by the head part is Qk time slots.

□

From Lemma 3.8 and 3.9 we conclude, that the read latency of the architecture is constantly Qk time slots. □

3.4.3.2 Assessment

This Section assesses the introduced read latency of the SPHSD. Therefore it first evaluates dependency on the parameter k. Then a comparison to the read latencies of other architectures is performed.

According to Theorem 3.4 the read latency of the SPHSD is **in time slots**

$$L_{SPHSD} = Qk$$

while Definition 3.2 defines a time slot **in seconds** as

$$\frac{b}{R} = \frac{B}{kR}$$

Expressed **in seconds** the read latency is

$$L_{SPHSD} = \frac{QB}{R} \tag{3.14}$$

As it can be seen, the read latency of the SPHSD is independent of k.

In the following the read latency of the SPHSD is compared to that of HSD and CFDS. For a simpler comparison we use as far as possible formulas that are cleared from effects that come from the authors' different assumptions on arriving data granularity (cf. Table 2.5 in Section 2.5.2.4).

The HSD has a read latency of $L_{HSD-ECQF} = QB/R$. Assuming DRAMs with $M = 8$ banks the read latency of CFDS is $L_{CFDS-ECQF-M8} = QB/R$. Concluding the read latency of all three systems is equal

$$L_{SPHSD} = L_{HSD-ECQF} = L_{CFDS-ECQF-M8}$$

Assuming future relevant DRAMs with $M = 16$ and $M = 32$ banks (cf. Section 3.4.1.2) read latency in CFDS decreases to $0.74\,QB/R$ and $0.61\,QB/R$, respectively, i. e., it outperforms the SPHSD.

Equivalently to the assessment of the tail buffer size, to make comparison fairer, here also additionally a CFDS is considered that suffers from less external DRAM fragmentation. We assume $P = 1.5Q$. Assuming DRAMs with $M = 8$, $M = 16$ and $M = 32$ banks the read latency of the CFDS is $1.5\,QB/R$, $1.11\,QB/R$, and $0.915\,QB/R$, respectively. Now the SPHSD slightly outperforms the CFDS for $M = 8$ and $M = 16$ banks.

The PHSD has a variable read latency and is therefore not considered.

Summarizing, for current and closer future DRAMs the SPHSD has the equal or shorter read latency compared to other systems. For farer future DRAMs with increased number of banks the CFDS slightly outperforms the SPHSD. If necessary, the system designer can decrease the read latency of the SPHSD to zero, by utilizing the MDQF head transferor algorithm proposed by Iyer *et al.* in [11] – however at the price of a significantly larger head buffer size.

3.4.4 DRAM Resources

One of the design targets for the SPHDS has been the efficient usage of the DRAM resources. This Section assesses the DRAM resource utilization in the SPHSD and compares this to that of other architectures.

Based on the design target to minimize DRAM resources Section 3.2 derived the necessary functions to be added to the SPHSD architecture. Accordingly, the SPHSD inherently meets this target.

Minimizing DRAM resources refers to three aspects: capacity, bandwidth, and data bus width. Explicitly, the SPHSD minimizes these by having the following properties

- Capacity: SPHSD does not suffer from internal or external fragmentation
- Bandwidth: SPHSD requires no speedup and does not suffers from internal fragmentation
- Data bus width: SPHSD utilizes banking

Table 3.1 faces the properties of the SPHSD and of the architectures presented in the related work Section.

	no internal fragmentation	no external fragmentation	DRAM banking	no speedup of DRAM bandwidth
HSD	✓	✓	–	✓
CFDS	–	–	✓	✓
PHSD	–	✓	–	–
SPHSD	✓	✓	✓	✓

Table 3.1: Summary of properties related to DRAM resource requirements

Concluding, the SPHSD architecture is the only that simultaneously features all properties and so utilizes DRAM most efficiently.

3.4.5 Summary of Results

Table 3.2 summarizes the metrics of the SPHSD: its tail buffer size, its head buffer size, and its read latency.

Metric	Formula	Source
Tail buffer size	$Q\frac{(k+1)}{2}$ [blocks]	Theorem 3.1
Head buffer size, MiRBD alg.	$Q(k+1)$ [blocks]	Theorem 3.3
Head buffer size, MaRBD alg.	$\approx Q\frac{(k+1)}{2}$ [blocks]	Section 3.4.2, not proven
Read latency	Qk [time slots]	Theorem 3.4

Table 3.2: Summary of metrics of the SPHSD; The size of a block is given in Eq. (3.5) while the duration of a time slot is given in Definition 3.2.

Consider that all values of all metrics of all architectures are theoretical bounds. In real implementation these bounds may cause roundings to meet real components' properties. For example, a DDR3 SDRAM chip has a burst length of 8 and at least 4 data pins what allows increasing the access granularity and total DRAM data bus width only in discrete steps.

Design of the SPHSD architecture was led by three main targets (cf. Section 3.1). The following summarizes the evaluation results with respect to these targets.

1. **Small SRAM size**
   With reasonable k the SPHSD requires a 47 % to 53 % smaller **tail buffer size** compared to other architectures (cf. Section 3.4.1.2). The **head buffer size** depends on the utilized algorithm. With the MiRBD algorithm and reasonable k the head buffer size is marginally larger or smaller depending on the compared architecture (cf. Section 3.4.2.2). With the MaRBD algorithm the reduction of the head buffer size is expected to be similar to the reduction of tail buffer size, i. e., approx. 50 %.

2. **Minimal DRAM resources**
   In contrast to related architectures the SPHSD features all necessary properties to minimize the DRAM resources: it eliminates any type of fragmentation, requires no speedup, and utilizes DRAM banking. Consequently, the SPHSD requires the theoretical minimal possible DRAM bandwidth of $2R$, no capacity overdimensioning, and a low DRAM data bus width.

3. **Deterministic bandwidth**
   The SPHSD inherently guarantees a deterministic bandwidth as it was designed and dimensioned accordingly.

3.5 Dimensioning Example

This Section illustrates by means of an example how SRAM and DRAM resources are dimensioned for a packet buffer that bases on the SPHSD architecture. The objective of this Section is to show the possibilities and degrees of freedom during the dimensioning process.

This Section is organized as follows. First it introduces the system requirements for the packet buffer. Second it presents the properties of the DRAM that should be utilized to implement the packet memory. Following, exemplarily three different alternatives are shown to access the DRAM which lead to different trade-offs. Finally it introduces two principally different dimensioning strategies and performs dimensioning for these.

System Requirements

The system to be dimensioned should suffice the following requirements

- Line rate: $R = 100\,\text{Gbps}$
- Capacity: $C = 2.5\,\text{Gbyte}$ ($C = R \cdot RTT$ with $RTT = 200\,\text{ms}$, cf. Section 2.2.4 on page 24)
- Number of flow queues: $Q = 1000$

We assume that the SRAM used to implement tail and head buffer has a short enough access time to accept and deliver the minimal size packets with line rate. Accordingly, knowledge about packet sizes is not required for this dimensioning example.

DRAM Properties

To keep costs low we chose to utilize DDR3 SDRAM to implement the packet memory. DDR3-1600K SDRAM [29] is currently one of the DDR3 SDRAMs delivering the highest bandwidths. In this dimensioning example we use a corresponding commercial DRAM chip from Micron: Part number MT41J256M4, speed bin -125 [65]. Table 3.3 lists its properties.

property	value	unit	comment
banks	8	#	
T	53.1	ns	mean random access time[3]
burst length	8	data words	
data pins/DRAM chip	4	#	
capacity/pin	0.25	Gbit	four times higher capacities also available
bus frequency	800	MHz	
t_{ck}	1.25	ns	clock cycle time
peak bandwidth	1.6	Gbps	theoretical value
data words/clk	2	#	DDR (double data rate)

Table 3.3: DDR3-1600K SDRAM chip properties (Micron part number MT41J256M4, speed bin -125)

In the following presented DRAM parameters and performance properties are partly estimated. This is sufficient for this example to provide close to reality calculations. Consider that exact dimensioning for a real implementation requires also exact parameters, as the packet buffer has to guarantee the bandwidth. Deriving these has to take into account the large number of timing constraints the DRAM has to meet.

The used DRAM chips feature 8 banks. To achieve a high data bus utilization we access the banks of the DRAM interleaved. However, its T_{RRD} and T_{FAW} timing constraints prohibit to perform one access to all 8 banks in a time period of T, i. e., to decrease the access time to $T/8$ (cf. Section 2.3.3 on page 32).

We estimated that up to 6 banks can be accessed in a time period of T when each access transfers just a single burst. Equivalently to footnote 3 we abstract thereby from penalties for changing the access type (read, write) as they can be alleviated by batching equal access types [31]. Figure 3.10 exemplarily illustrates interleaved access to 6 banks during a time period of T.

[3] A DDR3-1600K SDRAM device has an estimated mean random access time of $T = 53.1$ ns when the read/write ratio is 1 and each access transfers just a single burst. The mean value is required, because a read and a write access have different access times. This means in $2T = 106.2$ ns one read and one write access can be performed, while the read access takes 48.75 ns and the write access 57.45 ns. This calculation abstracts from penalties for changing the access type (read, write) as this can be alleviated to a large extent by batching equal access types [31], e. g., instead RWRWRWRW perform RRRRWWWW.

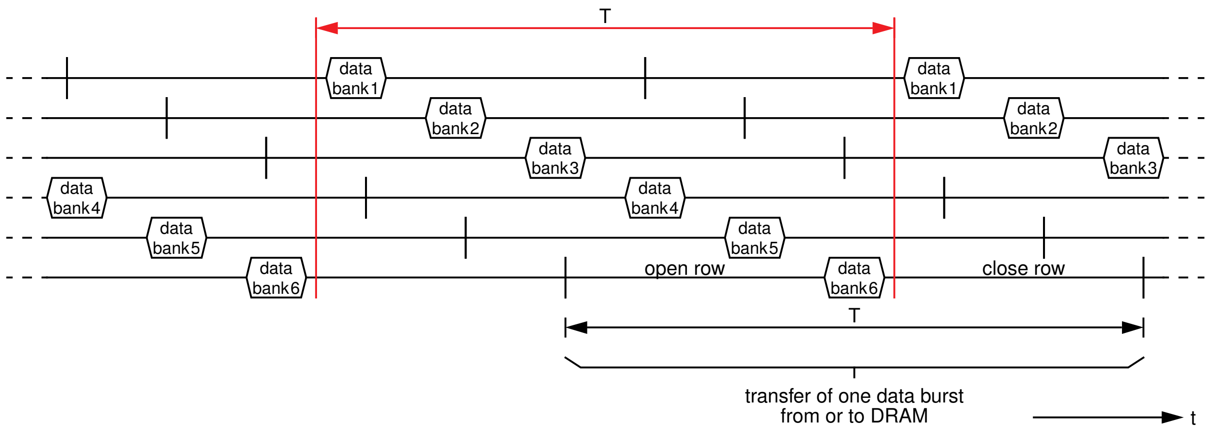


Figure 3.10: Interleaved access to 6 banks in round robin manner during a time period of T

DRAM Access Alternatives

There are many alternatives accessing the banks in this DRAM. Based on the previous estimated upper bound for bank interleaving we exemplarily show three reasonable alternatives which all lead to different trade-offs.

1. We can access the maximal number of 6 banks in a time period T while each access transfers one burst. To guarantee that this dense interleaving is consecutively possible the used banks have to be accessed in a strict round robin manner, i. e., 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, 1, etc. (cf. Figure 3.10). This leads to a data bus utilization of

$$\rho = \frac{\text{clock cycles used}}{\text{clock cycles of } T} = \frac{\text{banks} \cdot \frac{\text{burst length}}{2}}{\frac{T}{t_{ck}}} = \frac{6 \cdot \frac{8}{2}}{\frac{53.1\,\text{ns}}{1.25\,\text{ns}}} = 56.5\,\%$$

   As only 6 of 8 banks are used we waste 25 % of the available capacity. However, in reality this is not an issue as DDR3 SDRAM is extremely cheap and capacity of the final system will be anyway overdimensioned as it is harder to achieve the bandwidth requirement.

2. To be able to access all 8 banks we group them into 4 groups, e. g., (1,2), (3,4), (5,6), and (7,8). Now we perform each T one access to each **bank group** in strict round robin manner, i. e., group 1, group 2, group 3, group 4, group 1, etc. This means 4 banks are accessed during each interval of T, i. e., one of each group. The *per-flow round robin dispatcher* distributes the blocks of a flow in first order round robin over all groups and in second order round robin over all banks of a group. E. g., it dispatches consecutive blocks of a flow to the following banks: 1, 3, 5, 7, 2, 4, 6, 8, 1, 3, etc. However, performing just 4 accesses per time period T decreases data bus utilization by $\frac{1}{3}$ to

$$\rho = 37.7\,\%$$

3. To increase the poor data bus utilization provided by the previous alternative we can just transfer two data bursts per access. This increases the mean random access time to $T = 55.6\,\text{ns}$[4]. The data bus utilization increases to

$$\rho = \frac{\text{clock cycles used}}{\text{clock cycles of } T} = \frac{\text{groups} \cdot \frac{2\cdot \text{ burst length}}{2}}{\frac{T}{t_{ck}}} = \frac{4 \cdot \frac{2\cdot 8}{2}}{\frac{55.6\,\text{ns}}{1.25\,\text{ns}}} = 72\,\%$$

However, the required head and tail buffer sizes (SRAM) increase proportionally with T, i. e., by 4.7 %.

Providing the highest data bus utilization, we chose possibility 3 for all further calculations. At this point we can already derive the necessary width of the **DRAM data bus in pins**. To achieve the required capacity we need at least

$$p_{cap} = \left\lceil \frac{C}{\text{capacity/pin}} \right\rceil = \left\lceil \frac{2.5\,\text{Gbyte}}{0.25\,\text{Gbit/pin}} \right\rceil = 80\,\text{pins}$$

To achieve the total required bandwidth of $2R$ we need at least

$$p_{bw} = \left\lceil \frac{2 \cdot R}{\rho \cdot \text{peak bandwidth/pin}} \right\rceil = \left\lceil \frac{2 \cdot 100\,\text{Gbps}}{0.72 \cdot 1.6\,\text{Gbit/pin}} \right\rceil = 174\,\text{pins}$$

Consequently, the DRAM data bus has to be dimensioned to have at least

$$p = max(p_{cap}, p_{bw}) = 174\,\text{pins}$$

Dimensioning Strategy Alternatives

The formula $b = 2TR/k$ expresses the relation between the block size b and the degree of parallelism k. R is a fixed system parameter. T is defined by the utilized DRAM and the chosen access alternative. When dimensioning a packet buffer based on SPHSD we have now the freedom to either choose b and then calculate k or vice versa. In the following for both alternatives an exemplary dimensioning is performed.

Alternative 1: choose *b*, calculate *k* – Designing a packet buffer for an input line card it is favorable to choose the block size equal to the cell size of the switch fabric. E. g., choosing $b = \text{cell} = 64\,\text{byte}$ leads to

$$k = \left\lceil \frac{2TR}{b} \right\rceil = \left\lceil \frac{2 \cdot 55.6\,\text{ns} \cdot 100\,\text{Gbps}}{64\,\text{byte}} \right\rceil = 22$$

In the first step we have to group the available DRAM chips to **DRAM groups** to achieve the required access granularity of $b = 64\,\text{byte}$. All DRAMs in a DRAM group are operated in unison. With 8 DRAM chips with 4 pins each transferring two bursts per access (burst length 8) the access granularity is $8 \cdot 4 \cdot 2 \cdot 8\,\text{bit} = 64\,\text{byte}$.

[4]Calculation bases on the same assumptions as for the mean access time in Table 3.3. Transferring two bursts per access requires a read access time of 48.75 ns and a write access 62.45 ns, i. e., in mean 55.6 ns.

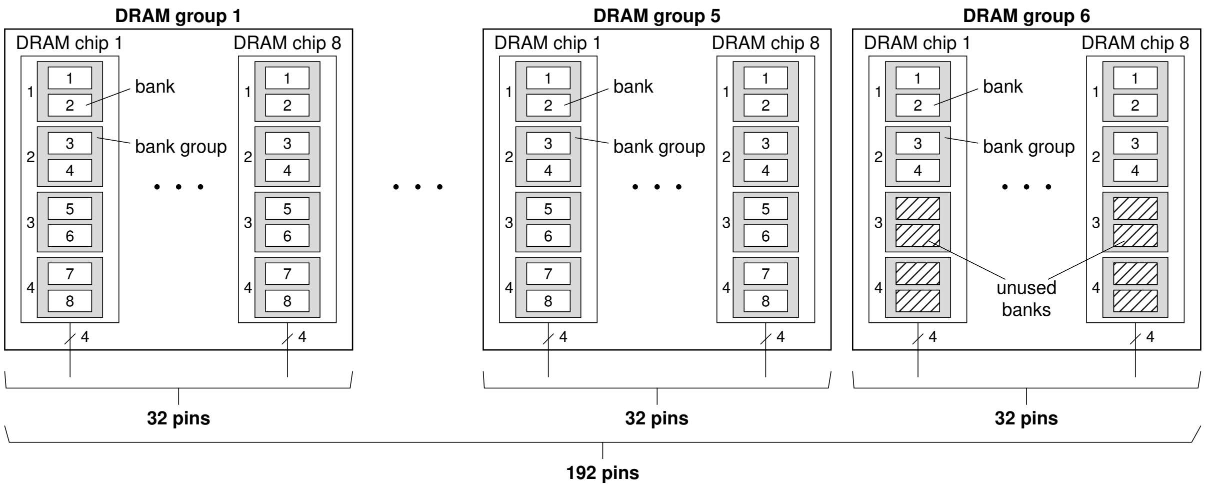


Figure 3.11: DRAM organization for dimensioning alternative *choose b*

In the second step we have to dimension the necessary number of DRAM groups to achieve k. With 4 accesses per DRAM group per T we need

$$\left\lceil \frac{k}{4} \right\rceil = \left\lceil \frac{22}{4} \right\rceil = \lceil 5.5 \rceil = 6 \text{ DRAM groups}$$

The 6th DRAM group will receive only 2 accesses per T to achieve exactly $k = 22$, i. e., it will utilize only 4 of 8 banks and will also deliver only 50 % of its bandwidth. Full utilization of the 6th DRAM group would lead on the one hand to a less filled tail buffer in average, but on the other hand to more DRAM queues to be managed in the tail buffer, without decreasing the upper bound of the tail buffer size. The 6 DRAM groups have in total $6 \cdot 32 = 192$ pins. The corresponding DRAM overdimensioning is $(192 - 174)/174 = 10.3\,\%$. The number of required pins is inherently correct due to used relation to calculate k: $k = \frac{2TR}{b}$:

$$\text{utilized DRAM groups} \cdot \text{pins of a DRAM group} = 5.5 \cdot 32\,\text{pins} = 176\,\text{pins} > 174\,\text{pins}$$

Figure 3.11 illustrates this DRAM organization.

As parameter k was rounded up, the real block size $b = 64\,$byte is marginally larger than the theoretical $b = \frac{2TR}{k} = 63.18\,$byte. Tail and head buffer size are calculated according to the formulas given in Theorem 3.1 and Theorem 3.3. Thereby we use the real block size to capture this marginal additional increase of the buffer sizes.

$$S_{tail-SPHSD} = Q\frac{(k+1)}{2}\text{ blocks} = 1000 \cdot \frac{(22+1)}{2} \cdot 64\,\text{byte} = 719\,\text{Kbyte}$$

$$S_{head-SPHSD-MiRBD} = Q(k+1)\text{ blocks} = 1000 \cdot (22+1) \cdot 64\,\text{byte} = 1438\,\text{Kbyte}$$

Alternative 2: choose *k*, calculate *b* – Designing a packet buffer where the block size b is not dictated we have an additional degree of freedom. Consequently, k can be used to minimize DRAM overdimensioning.

According to Section 3.4.1.2 a reasonable value for k is roughly in the range of 8 to 20. To determine which k lead to a small DRAM overdimensioning the author calculated

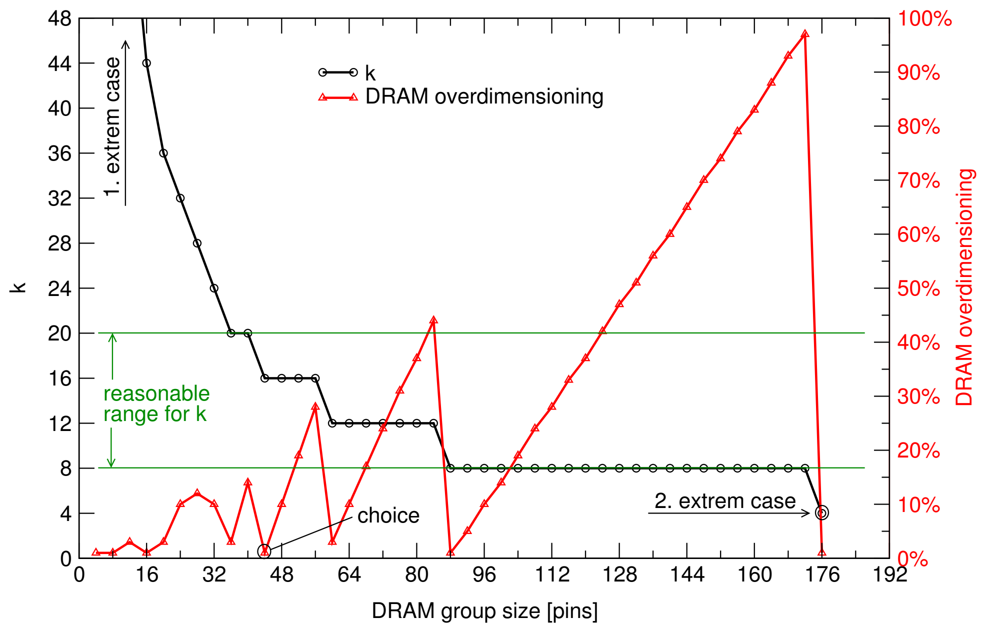


Figure 3.12: Parameter k and DRAM overdimensioning as function of the DRAM group size in pins, i. e., data pin count of one DRAM group

DRAM overdimensioning for all possible DRAM group sizes. The two extreme cases are having just one DRAM chip in each DRAM group and having all DRAM chips in just one DRAM group. Figure 3.12 shows parameter k and DRAM overdimensioning as a function of the DRAM group size in pins between these two extreme cases. As each DRAM group increases k by 4, the first extreme case leads to $k = \text{DRAM groups} \cdot 4 = \lceil \frac{174}{4} \rceil \cdot 4 = 176$ (not visible in Figure 3.12 due to scaling of the y-axis), while the second leads to $k = 4$ (cf. Figure 3.12).

Two effects contribute to DRAM overdimensioning. Firstly, when the required DRAM bus width is not an integer multiple of the data pins of a DRAM chip we have unavoidable overdimensioning. For example, we need here at least $\lceil p/4 \rceil = \lceil 174/4 \rceil = \lceil 43.5 \rceil =$ 44 DRAM chips. The half additional DRAM chip leads to $0.5/43.5 = 1.1\,\%$ unavoidable DRAM overdimensioning. The second effect is that the chosen DRAM group size can lead to an increased number of required DRAM chips, i. e., here more than the minimum of 44. This overdimensioning is avoidable. E. g., when we chose having 43 DRAM chips per DRAM group we need in total two DRAM groups. This leads to an overdimensioning of nearly 100 % (cf. Figure 3.12).

To minimize DRAM overdimensioning we chose from Figure 3.12 to have 44 pins per DRAM group, i. e., 11 DRAM chips per DRAM group and 4 DRAM groups in total. This leads to $k = 16$, a minimal possible DRAM overdimensioning of 1.1 %, and a block size of $b = 44 \cdot 2 \cdot 8\,\text{bit} = 88\,\text{byte}$. Figure 3.13 illustrates this DRAM organization.

Tail and head buffer size are calculated according to the formulas given in Theorem 3.1 and Theorem 3.3.

$$S_{tail-SPHSD} = Q\frac{(k+1)}{2}\text{ blocks} = 1000\frac{(16+1)}{2}\cdot 88\,\text{byte} = 730\,\text{Kbyte}$$

$$S_{head-SPHSD-MiRBD} = Q(k+1)\text{ blocks} = 1000\cdot(16+1)\cdot 88\,\text{byte} = 1461\,\text{Kbyte}$$

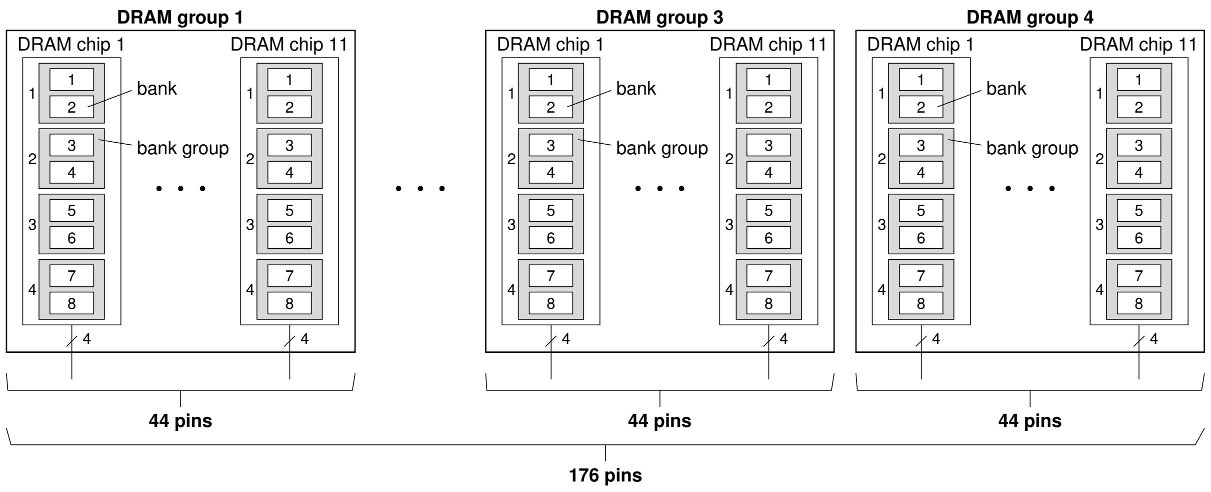


Figure 3.13: DRAM organization for dimensioning alternative *choose k*

4 Prototypical Implementation of a Packet Buffer for an Input Line Card

Packet buffering is a self-contained functionality. Consequently, to validate the SPHSD architecture introduced in Chapter 3 it is sufficient to implement and operate a corresponding packet buffer instead of a complete router. This Chapter presents a prototypical packet buffer implementation based on the SPHSD.

The SPHSD is applicable to any packet buffer that maintains a set of FIFO queues. Consequently, the author had to choose between implementing a packet buffer for an input or an output line card. He chose to implement a packet buffer for an input line card due to the following two reasons.

Firstly, in the SPHSD head and tail buffer sizes are proportional to the number of maintained queues Q. Consequently, the SPHSD is more suitable for packet buffers, where the number of queues maintained is moderate. This is the case on the input line card where the number of VOQs is in the range of hundreds to thousands [24, 27, 11].

Secondly, choosing the block size equal to the cell size of the switch fabric and letting the cells for the switch fabric contain aggregated packet data leads to two benefits: the required switch fabric bandwidth is significantly decreased *and* the head part implementation simplifies as assembly of cells from blocks is not necessary. As assembly is a simple functionality this choice does not decrease the prototype's validity.

To realize a hybrid packet buffer the designer needs SRAM and DRAM and a device that implements queue management. A packet buffer accepts and delivers packets with line rate in the fast path of a router. Consequently, in high-speed routers queue management is only feasible by implementation as dedicated logic circuit, i. e., in hardware.

The prototype is completely implemented in hardware and fully functional. Parts of it have been realized by student research projects supervised by the author [7, 8]. The author presented the main results regarding the tail part implementation on an international conference [1].

This Chapter is organized as follows. Section 4.1 shows the targets of the implementation. Section 4.2 introduces the hardware platform hosting the implementation. Section 4.3 presents the overall structure of the prototype. Section 4.4 and Section 4.5 show in detail the implementation of tail part and head part, respectively. Section 4.6 shows the validation of the prototype with respect to functionality, hardware resource requirements and supported line rate.

4.1 Targets

The main objectives of the prototypical packet buffer implementation are to **show**:

- **Functionality of the SPHSD –** Implementation in hardware respects all details of an architecture. It is therefore well suited to show that the SPHSD is operational.
- **Feasibility of the SPHSD architecture in hardware supporting high line rates –** Application of the SPHSD in a router is only possible, when the SPHSD can be implemented in hardware and implementation supports reasonable line rates.

4.2 Platform

This Section first gives an overview of devices suitable for dedicated logic circuit implementation. Then it introduces the utilized hardware platform and the digital systems design workflow.

Devices for Packet Buffer Implementation

FPGAs and ASICs are the two types of devices that enable dedicated logic circuit implementation at high frequencies. The following gives an overview of these two types of devices.

ASICs *(Application Specific Integrated Circuits)* are electronic devices that are designed for a specific task. Due to this high specialization, these devices can deliver the highest throughputs while requiring a very small chip area and minimal power consumption. However, the price for this is that manufactured ASICs cannot be changed any more and that the design and production process of a new device is very expensive and time consuming [90, 91]. In contrast to *full-custom* ASICs where implementation can be done on transistor level, also *semi-custom* ASICs and so called *Gate-Arrays* exist which base on a set of already designed and tested standard cells or a matrix of logic gates, respectively. The latter device variants drop price and development time by factors compared to full-custom ASICs.

FPGAs *(Field Programmable Gate Arrays)* are Programmable Logic Devices (PLDs). FPGAs not only allow configuring functionality of the device but doing this at any time and arbitrary often. This means system designers can implement on an FPGA with relatively small effort arbitrary digital systems. High flexibility at reasonable design frequencies made FPGAs very popular in the area of prototyping and even for products with smaller quantities. Due to the ability for reconfiguration, the achievable design frequencies are much lower than with ASICs.

The abbreviation FPGA is misleading, as an FPGA does not contain logic gates but Lookup Tables (LUTs). A LUT is a very small memory, which contains for every input combination the corresponding output value. Therewith, an LUT can efficiently realize any logic function. Further, programming an FPGA means configuring its elements (e. g., LUTs) and interconnecting them. Programming of modern SRAM-based FPGAs is done by transferring the according configuration bits to the FPGA.

The basic element of an FPGA is the configurable logic block, which contains one or more LUTs to realize logical functions as well as flip-flops. Modern FPGAs contain a matrix of 100,000 and more of these logic blocks, which can be connected by a large number of configurable interconnects to realize complex logical functions. The individual FPGA manufacturers use different architectures and names for the logic blocks used in their devices. Architecture of a logic block also evolves with the FPGA generations as manufacturers for example try to minimize the chip area required to realize logic functions.

Beside logic blocks, modern FPGAs also contain a large number of small SRAMs, i. e., embedded memory blocks. These memory blocks can be configured to operate at different port widths (e. g., $32K \times 1\,\text{bit}$, $16K \times 2\,\text{bit}$ up to $512 \times 64\,\text{bit}$), support one read and one write operation simultaneously (called dual-port memory[1]), and can be combined to larger memories [92]. The cutting-edge products of the two major FPGA manufacturers Altera and Xilinx provide currently $38.3\,\text{Mbit} = 4.7\,\text{Mbyte}$ [93] and $50\,\text{Mbit} = 6.1\,\text{Mbyte}$ [94] of on-chip SRAM, respectively.

High-speed serial transceivers for inter-chip communication and a large number of general-purpose I/Os that allow connecting high-speed SRAM and DRAM devices make modern FPGAs a good candidate to implement queue management for high-speed routers.

Concluding, the low price and the ability to run a new hardware design by just configuring the FPGA makes FPGAs much more appropriate for a prototypical implementation compared to ASICs.

Universal Hardware Platform

The Universal Hardware Platform (UHP) [95] of the Institute of Communication Networks and Computer Engineering at the University of Stuttgart is a hierarchical platform for prototyping. It consists of a main board (called UHP-1) that can connect to several FPGA based daughter boards (called UHP-2). Further, so called UHP-3 UHP-3 Mezzanin-Extension-Cards can connect to the UHP-2 boards.

To implement the prototype the author uses the current generation of the so-called UHP-2 board, which is available since 2005. The UHP-2 contains an Altera Stratix II FPGA of the type EP2S 60 F1020 C3 [96]. Additionally, the board contains many interfaces that are all connected to the central FPGA: DDR2 SDRAM slot, 2 electrical 1 Gbps Ethernet interfaces including the PHY-chip, a parallel interface (Centronics), a serial interface (Universal Asynchronous Receiver/Transmitter, UART), and 4 extension slots for UHP-3 Mezzanin-Extension-Cards for further extension. Figure 4.1 shows a photo of the UHP-2 board with two exemplarily plugged extension cards.

[1]Some modern FPGAs also contain so called *true dual-port memory* that is able to perform any two operations per clock cycle, e. g., two writes.

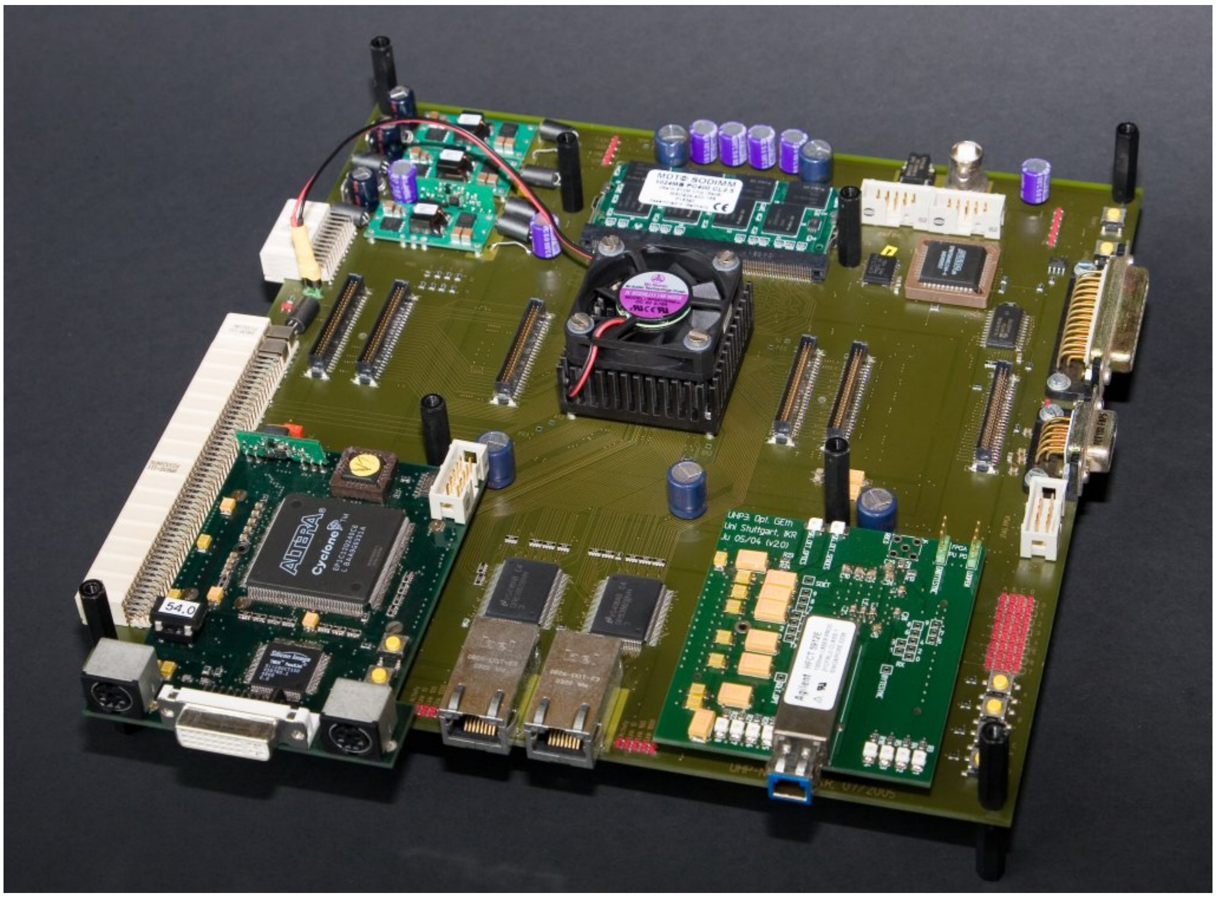


Figure 4.1: UHP-2 board

Digital Systems Design Workflow

The workflow for designing digital systems for FPGAs (but also ASICs) consists of five steps: description of the system, simulation, synthesis, place & route, and finally the configuration of the FPGA.

System Description – The designer describes the digital system with help of a Hardware Description Language (HDL). The two most popular HDLs are Verilog and VHDL (*VHSIC Hardware Description Language*, while VHSIC is the abbreviation for *Very High Speed Integrated Circuit*). These HDLs allow structural and behavioral description of a module. Description is mostly done on register transfer level, but use of logic gates is also supported. To design the current prototype the author used VHDL and the design environment *HDL Designer* [97] from Mentor Graphics.

Simulation – To validate the functionality of modules or even the complete system the designer simulates their behavior together with a test bed. The test bed, which is often implemented using an HDL too, stimulates the system under test by generating input signals and validates the output signals. The author used the simulation tool *Modelsim* [98] from Mentor Graphics to simulate the whole prototype design. Simulation is clock accurate and allows monitoring of any individual signal inside the system.

Synthesis – The synthesis step converts the system description into a netlist containing only elements that are available on the targeted FPGA or ASIC, i. e., in case of an FPGA these are LUTs, flip-flops, memory blocks, or other special blocks (e. g., CPU cores). A special software – the synthesis tool – performs the synthesis step by interpreting the HDL-code describing the system. The author used *Precision RTL Synthesis* [99] from Mentor Graphics to synthesize the VHDL code of the prototype.

Place & Route – In the *Place & Route* step a software efficiently places and interconnects the elements from the netlist on the target device. The challenge is thereby to keep delays low so that the design can operate at a high clock frequency. The software outputs a configuration file, which is required to configure the FPGA. Additionally, the software outputs detailed lists about timings and resource usage. For Place & Route the author used *Quartus II* [100], which is provided by the FPGA manufacturer Altera.

Configuration – The final step is configuring the FPGA. After configuration, the FPGA realizes the digital system described in the first step and is ready to use.

Management

For the operation of the prototype on an FPGA, it is desirable to have a tool that is able to set and monitor register values in the FPGA during operation. The author developed by help of colleagues and student research projects [101, 102, 103] such a tool called Management-System [104].

The Management-System enables to monitor register values (e. g., block counters) at any place in the design to detect malfunction. Further, we can re-configure modules that allow parameterization via register values by just modifying the according registers, i. e., without performing time consuming synthesis and place & route of the design. Finally, we also start and stop operation of the prototype with this tool.

The Management-System consists of two parts: modules that the designer adds to its HDL design and software that reads and writes the register values in the FGPA. The PC running the software and the FPGA communicate via an Ethernet connection.

4.3 Prototype Overview

This Section presents the overall prototype realizing a packet buffer for an input line card. It shows the prototype's structure, discusses design decisions, and introduces basic design properties.

4.3.1 Overall System

Figure 4.2 shows the overall block diagram of the prototype. The **proposed hybrid memory architecture (SPHSD)** (cf. Chapter 3) in the middle of the figure is the design under test.

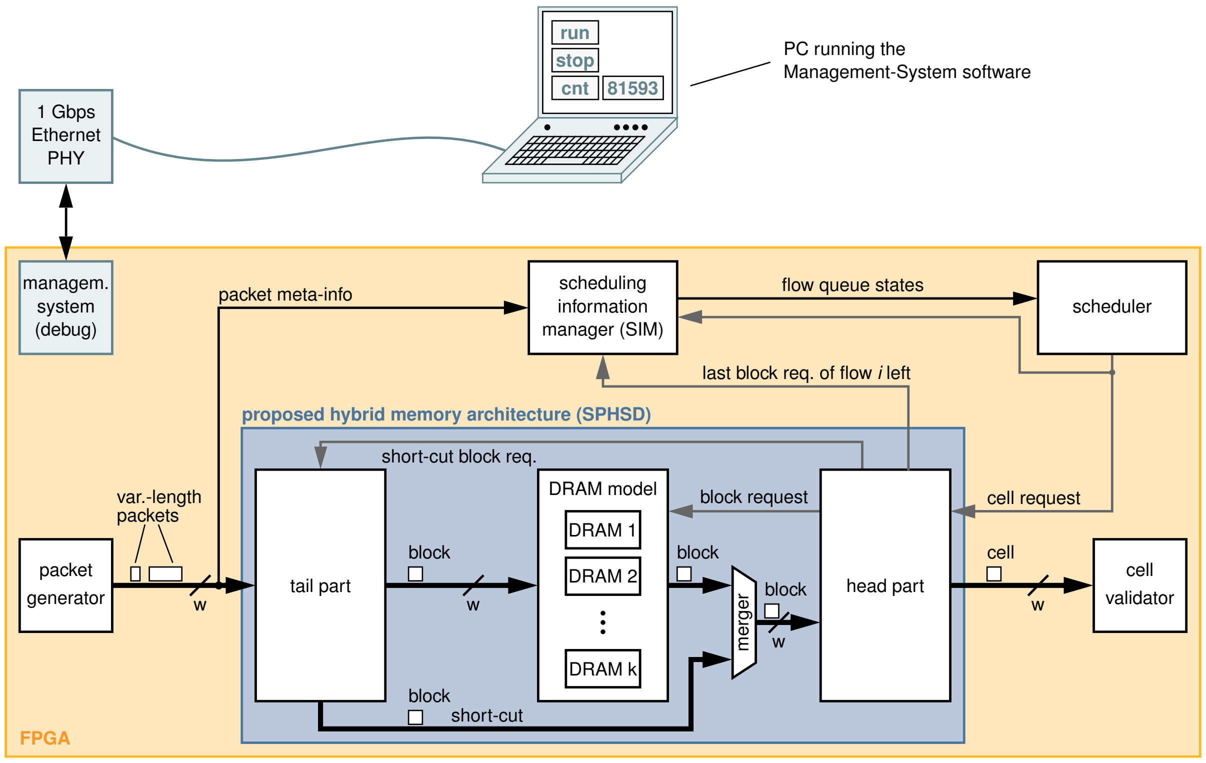


Figure 4.2: Overview of the complete prototype

The tester, consisting of packet generator, cell validator, scheduler, and Scheduling Information Manager (SIM) surrounds the SPHSD. Tester and SPHSD are realized on a single FPGA. Finally, a PC running the Management-System software connects to the FPGA via an Ethernet link.

The SPHSD receives at its input variable-length packets and delivers constant-size cells at its output. As already mentioned on page 103, we let the cells contain aggregated packet data to minimize the bandwidth of the subsequent switch fabric. Further, we choose the internal block size of the SPHSD equal to the cell size used by the switch fabric. This simplifies the head part implementation as assembly of cells from blocks is not necessary. As assembly is a simple functionality this choice does not decrease the prototype's validity. The switch fabric itself is not essential for the validation of the SPHSD and is therefore omitted.

The following paragraphs describe the functionality of the individual building blocks of the prototype.

Scheduler and Scheduling Information Manager

The **scheduler** represents the switch fabric scheduler. It requests cells from the flow queues maintained by the SPHSD. As the explicit scheduling algorithm used is nonessential, the scheduler implements a round robin scheduling algorithm. To be able to schedule, the scheduler has to know the state of the flow queues. The SPHSD cannot provide this information as none of its three main parts (tail, head, DRAMs) has a global view of the flow queues' states. Consequently, an additional module – the **scheduling information manager** (SIM) – maintains the flow queue state information required for scheduling. To keep the status up-to-date the SIM receives the meta-information (i. e., packet length and flow-id) of every packet entering the SPHSD and every cell request generated by the scheduler.

How much flow queue state information the SIM has to hold depends on the scenario and the algorithm the scheduler implements. In this scenario we assume that the scheduler has to know only the length of the individual flow queues in byte to schedule blocks. Accordingly, the SIM in the prototype maintains just one byte-counter per flow queue and is therefore simple[2]. We assume that the packet lengths are stored along with the packets, as the output buffer needs this information to reassemble the packets.

Aggregation and potential queuing of block requests in the head part lead to following problem. Assume the case when the scheduler requests the last cell of flow *i* that is currently available in the SPHSD. At this time, with a high probability the corresponding block is non-full and therefore resides in the tail part. The head part cannot process the corresponding block request immediately, but after a queuing delay. Meanwhile, the tail part may receive additional packets for flow *i* and this block may get full. Concluding, upon generating a cell request for a corresponding non-full block, the scheduler cannot know if the block will be full or not at the time of delivery. By this, the state information in the SIM and the state in the SPHSD may get out-of-sync.

The following observation makes it simple to solve the problem: at any time, only the last requested block of a flow can be potentially non-full. Explicitly, the head part signals to the SIM, when from the block requests of a flow currently residing in the head part, the last leaves towards DRAMs or tail part, i. e., the *last block request of flow i left* (cf. Figure 4.2). Knowing the exact time when a block request leaves the head part the SIM can take corrective access to the byte-counter of the corresponding flow if necessary.

Packet Source and Cell Sink

The contents of the buffered packets are without any relevance when validating SPHSD functionality. Consequently, one can use an on-chip packet generator, which has two main advantages over real network traffic. Firstly, the packet generator can easily generate packets with full line rate. Secondly, a corresponding cell validator can verify the cells delivered by the SPHSD on the fly. Validation of the cells' content is simple, as one knows the content of the generated packets. Here the individual bytes of a packet contain the byte count of the corresponding flow, i. e., 0, 1, 2, ..., 254, 255, 0, 1, etc. Two pseudo-random number generators[3] in the packet generator deliver for each packet a *length* and a *flow-id*. Pseudo-random behavior is important as it allows reproducing results and errors.

[2]The SIM is more challenging when it has to hold the individual packet lengths. Iyer proposes in [11] an approach that allows to extract queue state information from a hybrid memory architecture on the fly. Applying this approach, the SIM can be omitted.

[3]The pseudo-random number generation was implemented with help of 32 bit *linear feedback shift registers* with maximal period.

We chose this simple packet content to simplify debugging during simulations. Despite simplicity, the probability to miss an error is practically zero. This has three reasons. Firstly, the probability that two identical blocks exist in the SPHSD system is small due to the presence of randomness and short-cut. Secondly, in our implementation the blocks always carry meta-information, except when they are stored in the tail buffer. Meta-information prohibits undetected mix-up of two blocks. Thirdly, the tail buffer is a simple dual-port memory that sequentially receives and delivers blocks. Consequently, for an undetected mix-up of two blocks two errors have to happen, which affect exactly two equal blocks.

Management

For management and debugging purposes we use the Management-System introduced in Section 4.2. The PC running the Management-System software connects to the FPGA via a 1 Gbps Ethernet link. Examples for the usage are start or stop of testing, setting the starting condition of the pseudo-random packet generators, monitoring of numerous block and packet counters to locate failures in the design, etc.

SPHSD

From the two head transferor algorithms introduced in Section 3.3.4 we choose to implement the *MiRBD algorithm* as for this the head buffer size has been proven.

The SPHSD accesses the DRAMs systematically, i. e., it accesses each of the k DRAMs once per random access time T and it performs access to the k DRAMs in strict round robin manner (cf. DRAM access alternatives in Section 3.5). Consequently, we can use a simple DRAM model that reflects these properties to implement the *parallel DRAMs*. Beside simplicity, the model has a main advantage over DRAM devices: it can be configured to provide any k and any T.

The DRAM model does not need to account for the refresh cycles necessary in DRAM devices due to the following two reasons. Firstly, DRAM dimensioning is performed based on the net bandwidth of a DRAM device. Secondly, the DRAM controller can compensate the temporal loss of bandwidth due to a refresh cycle with help of a small buffer. For example, assume that the DRAM controller has to buffer two blocks due to the occurrence of a refresh cycle. When the next refresh cycle is necessary, this buffer is empty as the DRAM device caught up in the mean time. In other words, the random access time T used to dimension the SPHSD is slightly larger than the real random access time of the DRAM device in order to account for the refresh cycles.

The author implemented the simple DRAM model with on-chip SRAM available on the FPGA. Further, each DRAM in the model already supports a set of Q queues. This supersedes implementation of a memory manager to manage the flow queues in the DRAM. As queue management with linked lists (cf. Section 2.2.3) is state of the art, this does not impact the results regarding the feasibility of the SPHSD.

The author implemented tail and head buffer with on-chip dual-port SRAM. The dual-port property cuts its access time by half, what enables implementation for very high line rates on an FPGA.

During implementation, the focus was set on achieving full design functionality and keeping it highly configurable. Optimization on throughput, i. e., design frequency, would require a second implementation considering the lessons learned from the first. Throughout this Chapter, the author will point out the suboptimal design decisions that limit design frequency and propose throughput optimized alternatives.

4.3.2 Basic Design Properties

This Section introduces the basic properties of the prototype implementation. This means, it discusses the choice of the bus width, the design frequency, the block size, etc. Finally, it introduces the control and meta-information required for the packets, blocks, and cells in the design.

Dimensioning of Bus Width and Frequency

The input and output bus of the SPHSD as well as all its internal busses have a width of w (cf. Figure 4.2). The size of w is an integer multiple of a byte, as the size of packets is usually byte granular. w is a configurable design parameter.

A module transfers a packet on the bus as a sequence of N words, while the last word may be utilized only partly. Further, a bus word always contains just data of a single packet.

The input bus provides a maximal data rate of $R_{gross} = f \cdot w$, where f is the operating frequency. The larger w is the lower is the required frequency f to achieve a data rate R, i. e., this allows a trade-off between w and f. However, since a bus word always carries just data of a single packet for the minimal frequency f_{min} it has to hold

$$f_{min} \geq r_{max} = \frac{R}{P_{min} + G_{min}} \tag{4.1}$$

r_{max} is the maximal packet rate. Its value depends on the minimal allowed packet size P_{min}. G_{min} is the minimal inter-framing gap of the used network technology.

Choosing $w = P_{min}$ is by far not the optimal solution with respect to the operating frequency. To carry a stream of packets of the size $P_{min} + 1$ byte a frequency of nearly $2 \cdot f_{min}$ is necessary to achieve the same net data rate. This phenomenon was referred to in the previous Chapters as 65-byte-problem.

Now a formula is derived to calculate w at a given frequency. As the objective is to keep f small, here **only the case** $w \geq P_{min}$ **is considered**. As mentioned before, the worst-case packet size is $P_{wc} = w + 1$ byte as this requires a frequency of

$$f \geq 2 \cdot r_{wc} \tag{4.2}$$

while r_{wc} is the packet rate at the worst-case packet size P_{wc}. Packet rate r derives from the line rate R and the packet size P

$$r = \frac{R}{P + G_{min}} \tag{4.3}$$

From Eq. (4.2) and Eq. (4.3) follows for the case $w \geq P_{min}$

$$f \geq \frac{2R}{w + 1\,\text{byte} + G_{min}} \qquad \Leftrightarrow \qquad w \geq \frac{2R}{f} - 1\,\text{byte} - G_{min} \tag{4.4}$$

To give an explicit example, Ethernet has a $P_{min} = 64\,\text{byte}$ and a $G_{min} = 20\,\text{byte}$. Accordingly, a 100 Gbps Ethernet line card requires an $f_{min} = 148.8\,\text{MHz}$ at a bus width of $w = 1176\,\text{bit} = 147\,\text{byte}$. For an exemplary bus width of $w = 512\,\text{bit} = 64\,\text{byte}$ the required frequency is $f = 294.1\,\text{MHz}$.

System Clock Frequency

It is easier to achieve a deterministic bandwidth in a system which contains no asynchronous clock domain crossings. Consequently, we choose the operating frequency f of the input bus to be the operating frequency of nearly the complete system, called **system clock frequency**. As it will be shown later, some memories need to run at twice the system clock frequency. However, this introduces synchronous clock domain crossings only.

Block Size

The on-chip memories used to implement head and tail buffer feature an input and output port width of w, so they can operate at the same frequency as the rest of the design. One of the design objectives in Chapter 3 was to minimize the required SRAM sizes by eliminating any internal fragmentation. Consequently, the block size b has to be an integer multiple of words w.

The prototype uses a block size that is equal to the bus width, i. e., $b = w$. This choice allows realizing the smallest tail buffer size (cf. Section 3.4.1). Any larger block size $b = n \cdot w$ with $n \geq 2$ does neither change the architecture properties nor complexity of the implementation as long as each module transfers a block as a whole to the next module, i. e., in n consecutive clock cycles.

Time Slot

A *time slot* is the time to receive b byte at line rate R (cf. Definition 3.2, page 71). In a digital system with a block size of $b = w$ the duration of a time slot is in the range of $[1;2)$ system clock cycles. This is true, when using a system clock frequency according to Eq. (4.4), i. e., the system clock frequency is not overdimensioned.

The minimal value of 1 means, that on the input bus b byte can arrive every clock cycle. This is only possible when there is no fragmentation on the input bus, i. e., the arriving packets' size is always an integer multiple of the block size.

The maximal value of nearly 2 means, that it takes in average nearly two clock cycles two receive b byte on the input bus. This case occurs, when the designer chooses $w = P_{\min}$ (cf. bus width dimensioning prior in this Section, page 111).

In the SPHSD, the time budget to process a full block b is one time slot. This means, in the first case the system has to finish processing one full block each clock cycle, while in the second case it has nearly two clock cycles for the same task. Consider, that not the absolute duration of a time slot changes, but the system clock frequency.

To allow this highly configurable prototype to operate correctly with any given configuration, the author designs it to be always able to finish processing one aggregated block per system clock cycle. This corresponds to a worst-case assumption regarding the available processing time.

Control- and Meta-Information

Each data element (word of a packet, block, block request, cell, and cell request) has associated control and meta-information. Control-information signals whether an element is valid or not and on the input bus additionally if it is the first or the last word of a packet. Meta-information contains further information about an element that is necessary for its processing and association. Table 4.1 lists the meta-information of all data elements.

4.4 Tail Part

This Section introduces the implementation of the tail part. To simplify reading of this Chapter the author chooses a top to bottom approach. Section 4.4.1 introduces the block diagram of the tail part and describes its functionality. Subsequent Sections describe the individual modules of the tail part in more detail and discuss design alternatives.

4.4.1 Block Diagram

This Section presents the block diagram of the SPHSD's tail part. Therefore, it introduces the functional requirements of tail part and derives a block diagram that fulfills these. Finally, it describes the functionality of the individual modules.

The tail part has five main functions. Table 4.2 lists these. Further, the tail part has to fulfill all functions in a way, that it guarantees deterministic bandwidth. This means explicitly, it has to be able to accept packet words on its input and deliver blocks to the DRAMs continuously, i. e., without any interrupt.

The functions can be directly mapped to one or more modules of the tail part. Table 4.2 shows this mapping. Figure 4.3 depicts the block diagram of the tail part. A design alternative, which would slightly change the structure, is discussed in Section 4.4.2. The block diagram is explained in the following.

meta-inforamtion	range of values	comment
word of a packet		
flow-id	$[0,\ Q-1]$	Flow, to which the packet belongs. This is the classification result determined by a not considered preceding network processor.
packet length	$[0,\ P_{max}]$	This is the length of the packet in byte. The number of valid bytes in the last word of a packet follows from the packet length.
block		
flow-id	$[0,\ Q-1]$	Flow queue, to which the block belongs. This is equal to the flow number, as one flow queue is maintained per flow.
dram-id	$[0,\ k-1]$	DRAM, to which the block belongs. The dispatcher in the tail part assigns the value.
valid-bytes	$[0,\ w]$	The number of valid byte in the block. Information is required, as short-cut blocks can be non-full.
block request, short-cut block request		
flow-id	$[0,\ Q-1]$	Flow queue, from which the block is requested. This is identical to the flow-id, as one flow queue is maintained per flow.
dram-id	$[0,\ k-1]$	DRAM, from which the block is requested. The requester in the head part assigns the value.
cell request		
flow-id	$[0,\ Q-1]$	Flow, from which the cell is requested.
cell		
flow-id	$[0,\ Q-1]$	Flow, from which the cell is requested.
valid-bytes	$[0,\ w]$	The number of valid bytes in the cell. Information is required, as due to short-cut cells (blocks) may be non-full.

Table 4.1: Meta-information of the different data elements in the prototype

	function	**tail part module implementing it**
1.	aggregation of packets to blocks	$\rightarrow$ aggregation module
2.	dispatching of blocks to DRAMs	$\rightarrow$ dispatcher module
3.	buffering of blocks	$\rightarrow$ tail buffer
	(queues share buffer dynamically)	$\rightarrow$ dynamic memory manager module
4.	triggering block transfers to DRAM	$\rightarrow$ tail transferor module
5.	processing of short-cut requests	$\rightarrow$ short-cut module
		$\rightarrow$ short-cut reorder buffer

Table 4.2: Functions of the tail part and their mapping to modules

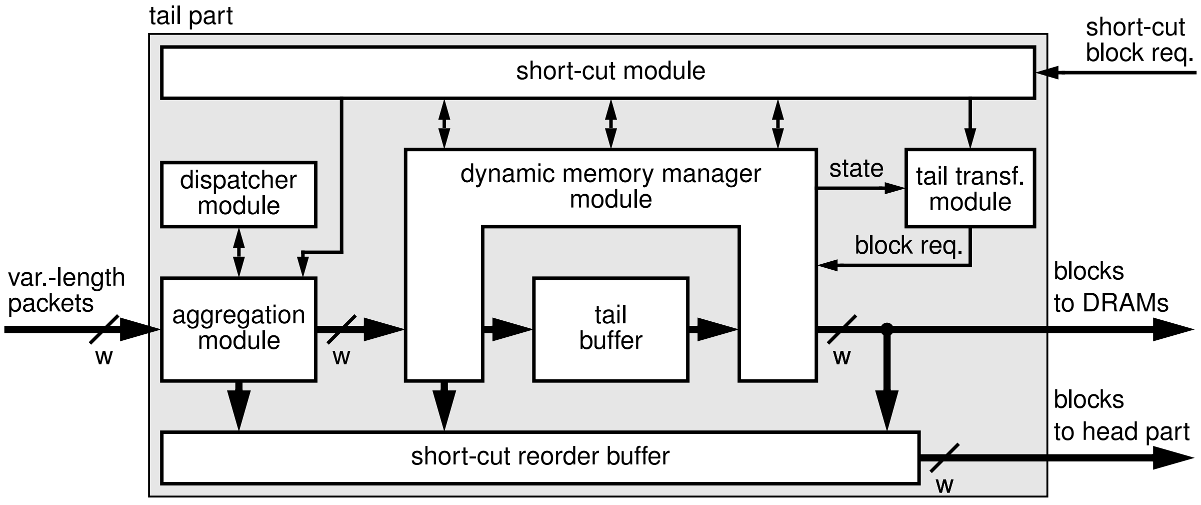


Figure 4.3: Tail part

From left to right in Figure 4.3 the **aggregation module** segments incoming packets and aggregates them to blocks per flow. To hold non-full blocks it utilizes a small memory. Whenever it starts aggregating a completely new block, it retrieves the corresponding target dram-id from the **dispatcher module**. When a block is full, the aggregation module forwards it to the **Dynamic Memory Manager module** (DMM module). The DMM module maintains the DRAM queues in the **tail buffer** and allows them sharing it dynamically. Further, it provides the state information of the individual DRAM queues to the **tail transferor module**. The tail transferor module checks the state of each DRAM queue periodically, i. e., every *k* time slots. Whenever it finds a DRAM queue non-empty, it triggers the transmission of a block from this DRAM queue to DRAM. Therefore, it sends a block request to the DMM module, which delivers the block after a few clock cycles to the DRAMs.

The **short-cut module** processes incoming short-cut block requests generated by the head part. A short-cut block request addresses the headmost block of a flow dispatched to a specific DRAM. The addressed block can be anywhere in the tail part along the data path. Upon arrival of a short-cut block request, the short-cut module searches in parallel the block's location. Then it triggers the corresponding module to forward the block to the **short-cut reorder buffer**. This reorder buffer is a small buffer, which can hold a few blocks. It is necessary, as the head buffer requires in-order delivery but retrieval time of a block to be cut short depends on its location along the data path. After reordering and with a constant read latency the reorder buffer forwards the short-cut blocks towards the head part.

The following Sections discuss the realization of the individual modules in more detail. Thereby they explicitly point out how short-cut functionality and the deterministic property influence implementation.

4.4.2 Aggregation Module

This Section discusses the aggregation functionality performed in the tail part and consists of three paragraphs. The first paragraph introduces and compares two implementation alternatives

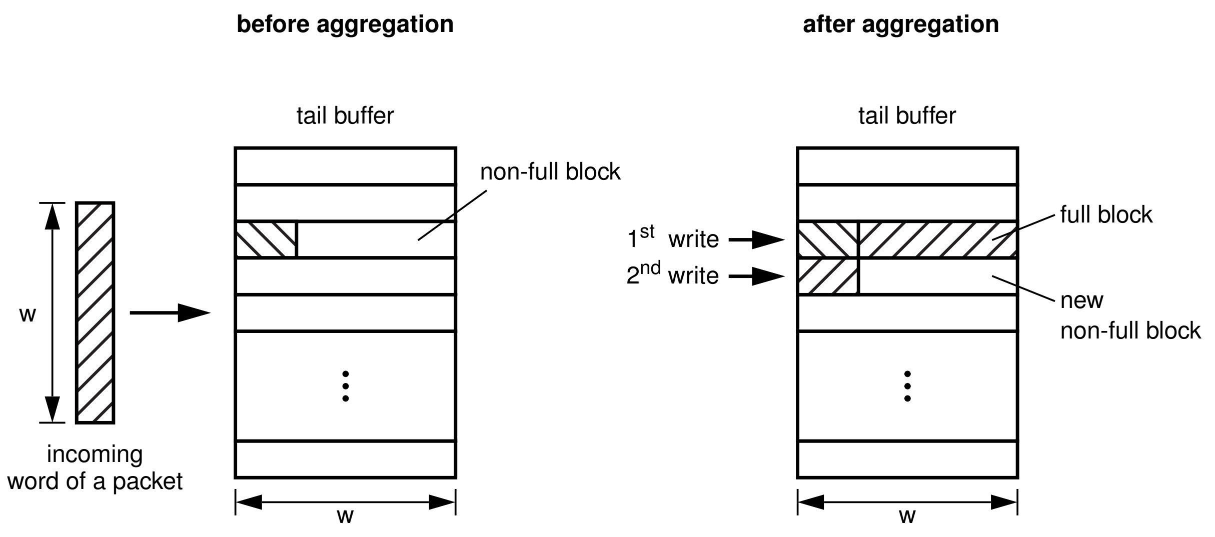


Figure 4.4: Aggregation in the tail buffer: requires two write accesses to the tail buffer per incoming word

where to realize aggregation in the tail part. Then it concludes that a separate aggregation module is more reasonable as it simplifies implementation. The second paragraph shows the influence of the short-cut functionality on the aggregation module while the third presents its block diagram.

Alternatives Implementing Aggregation in the Tail Part

Two basic alternatives exist to implement the aggregation functionality in the tail part: performing aggregation in the tail buffer or in a separate module containing memory dedicated to aggregation. These alternatives are discussed in the following.

Aggregation in the tail buffer – Performing aggregation in the tail buffer means that the tail buffer also holds non-full blocks. Consequently, an incoming word leads to two write accesses to the tail buffer: one write access to complete a non-full word already residing in the tail buffer and another to store the remaining data of the incoming word. Figure 4.4 illustrates this by showing the tail buffer before and after the two write accesses. This alternative has two consequences. Firstly, the DMM module has to manage aggregation additionally. Secondly, today's FPGAs do not support memories with two write ports and one read port. Emulating a second write with dual-port memory (cf. Section 4.2) requires either running the tail buffer at double frequency compared to the system frequency *or* doubling the tail buffer. However, utilizing an ASIC that contains on-chip memory with two write ports is also an option.

Aggregation in a separate module – Performing aggregation in a separate module, i. e., in the *aggregation module*, separates the tasks aggregation *and* buffering in overload situations. The aggregation module performs only aggregation and utilizes therefore a dedicated memory. This contains only non-full blocks. The tail buffer performs buffering of blocks in overload situations and contains only full blocks. Separation reduces complexity by modularization without requiring more memory than the first alternative. The latter is true as the memory required for aggregation (Q blocks) is just relocated from tail buffer to the aggregation module.

We chose to implement the second alternative having a dedicated aggregation module. This has two reasons. Firstly, this choice reduces implementation complexity. Secondly, a design target of the SPHSD is reduction of the tail buffer size as this is a critical resource. Emulation of a second write port would make it even more critical.

Influence of Short-Cut Functionality

To suffice the deterministic property of the SPHSD architecture the aggregation module has to accept every clock cycle a new incoming word **and** a short-cut request from the short-cut module simultaneously.

For aggregation of incoming words, the aggregation module accesses the aggregation memory twice: reading a non-full block and writing a new non-full block. To process the short-cut request the aggregation memory has to support an additional read per clock cycle, i. e., to read the non-full block to be short cut. In total the aggregation memory has to support two read and one write access per clock cycle (2×R, 1×W).

Today's FPGAs do not support memories with two read ports and one write port. As in the case where an additional write port was required (cf. page 116) we have three options: emulation by doubling frequency or memory itself, or utilization of an ASIC instead of an FPGA. The first two options are a trade-off between clock frequency and memory size. To keep design complexity of the prototype low we choose to double the aggregation memory.

Block Diagram

Figure 4.5 depicts the block diagram of the aggregation module. It consists of three types of modules: memories that hold aggregation state information, a merger module that performs the actual aggregation, and a control module that coordinates aggregation.

The aggregation state information of a flow consist of a non-full data block plus meta-information (cf. Table 4.1), i. e., valid-bytes, flow-id, and dram-id. The aggregation module holds maximally one non-full block per flow, i. e., Q non-full blocks in total. This allows assigning each flow a static memory location (address), i. e., each memory consists of Q memory words. Static assignment reduces implementation complexity without introducing drawbacks. Consequently, this supersedes explicit storage of the flow-id. The aggregation module holds the remaining aggregation state information in separate memories: **aggregation memory**, **valid-bytes memory**, and **dram-id memory**.

As already introduced, the **aggregation memory** requires two read ports and one write port (2×R, 1×W). The **valid-bytes memory** also has to fulfill this requirement, as the valid-bytes information is always necessary to process a non-full data block. Upon a short-cut request, the corresponding entry in the valid-bytes memory must be set to zero, i. e., reset. This requires an additional write port for the valid-bytes memory, i. e., in total 2×R, 2×W. We implement

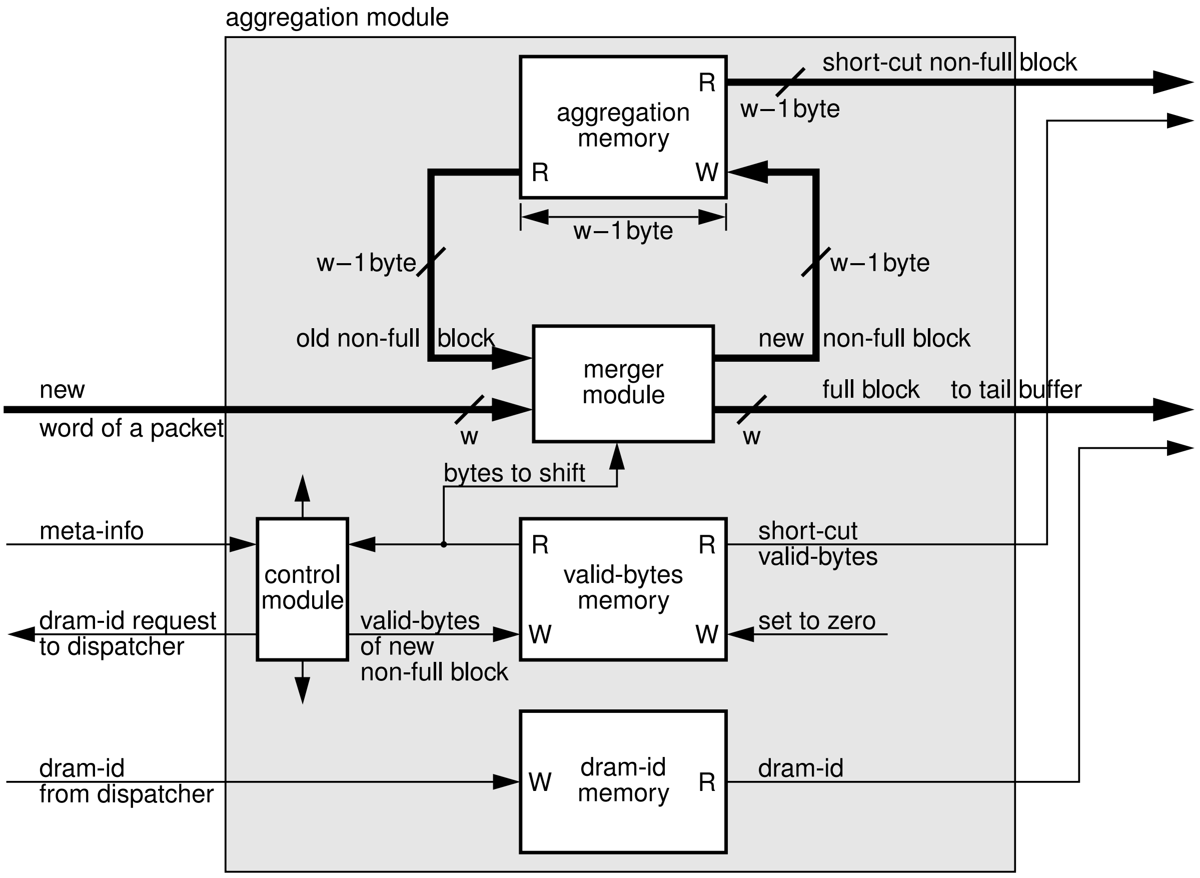


Figure 4.5: Aggregation module

this similarly to the aggregation memory while we additionally utilize a Q bit wide register. The register allows marking individual entries of the memory as reset by storing a '1' at the corresponding bit position. Finally, the **dram-id memory** stores the dram-id of the non-full block. It requires just one read and one write port, since short-cut block requests already contain the dram-id themselves.

The **merger module** performs the actual segmentation and aggregation. It receives a new incoming word of a packet and the old non-full data block and merges them to a full block and a new non-full block. Figure 4.6 visualizes this functionality. However, as the last word of a packet may contain just a small number of valid-bytes, two corner cases exist. Firstly, when the last word exactly completes aggregation of a block the merger generates only a full block. Secondly, when the valid-bytes of the word are not enough to complete aggregation of a block the merger generates just a new non-full block.

The merger module implements shifting of the new packet word by help of a barrel shifter [105]. The barrel shifter itself realizes shifting with help of multiplexers. The number of required multiplexers depends on the word width and the shifting range. These are here w and $[0, w-1]$, respectively. In a configuration supporting high line rates at moderate frequencies w is very large (cf. Section 4.3.2). Consequently, in such a configuration the merger module is resource-intensive. The signal propagation delay through the merger module increases with the shifting

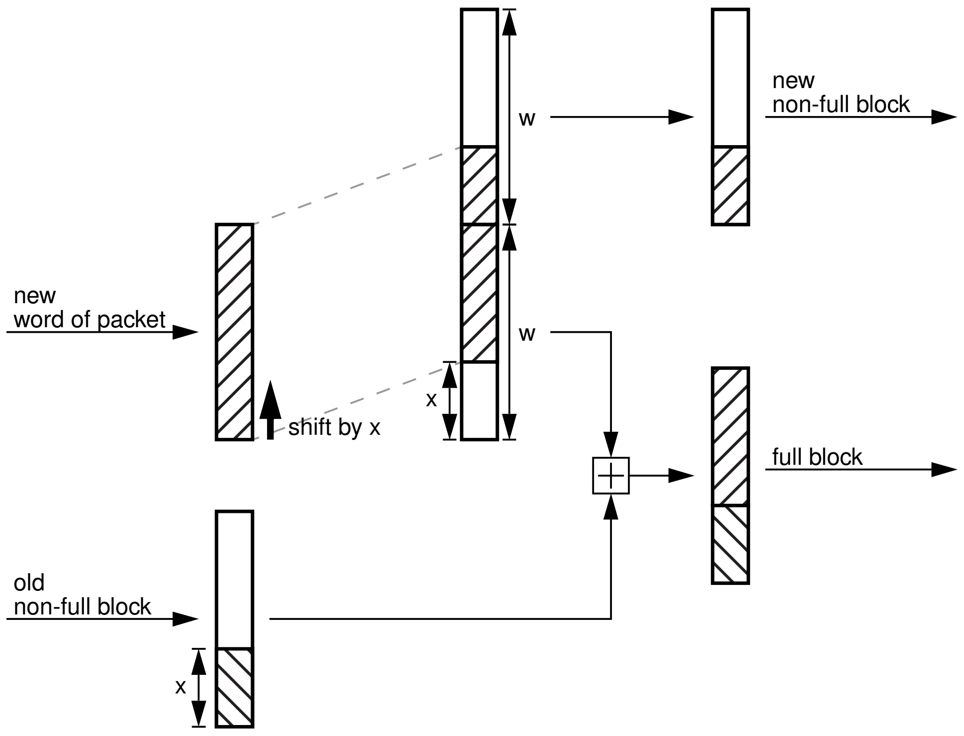


Figure 4.6: Aggregation performed in the merger module

range. However, it can be pipelined easily what enables high operating frequencies. For the sake of simplicity we implemented a combinatorial merger module for the prototype.

The **control module** coordinates aggregation. It has three different tasks. Firstly, it requests from the dispatcher a new dram-id whenever it starts the aggregation of a new block. Secondly, it calculates the valid-bytes of the current word of a packet from the packet length. Thirdly, it controls write accesses to the memories.

Summarizing, the aggregation module implements one of the most crucial functions of the proposed SPHSD architecture. The module is of medium complexity and pipelineable, what enables high operating frequencies. The largest impact on its implementation has the short-cut functionality, which requires additional read and write accesses to some of the internal memories.

4.4.3 Dispatcher Module

This Section presents the dispatcher module and is organized in two paragraphs. The first shows the module's task and placement in the tail part. The second introduces its block diagram.

Task and Module Placement in the Tail Part

The dispatcher module implements the *per-flow round robin dispatcher* introduced in Section 3.3.3. Its task is assigning each block a dram-id according to the per-flow round robin algorithm.

The head part of the SPHSD implements a requester module with similar functionality. When the head part generates a block request it cannot know if the corresponding block will be full or non-full at the time of its retrieval. Consequently, to keep dispatching in the tail part and requesting in the head part synchronized, both assign every non-zero block a dram-id. Therewith, the head part always knows for each flow which DRAM contains the next block.

For the placement of the dispatcher module two alternatives exist: right behind the aggregation module or in parallel to it.

Placing it **right behind the aggregation module** has the pro that the aggregation module does not need to know about dram-ids and consequently does not need a dram-id memory. The con is that the dispatcher module has to be able to process two dram-id requests per clock cycle: one for the block that currently finished aggregation and one for the non-full block that is short-cut simultaneously.

Placing it **in parallel to the aggregation module** has the pro that it has to processes maximally one dram-id request per clock cycle, since the aggregation module generates at most one new block (full or non-full) per clock cycle. The con is that the aggregation module has to implement a dram-id memory to store the dram-ids of the non-full blocks. This dram-id memory is relatively small and needs to support just one read and one write access per clock cycle.

The two placement alternatives are a trade-off between chip area (i. e., memory) and module operating frequency. For the prototype we choose the second alternative as it relaxes frequency requirements.

Block Diagram

Figure 4.7 depicts the block diagram of the dispatcher module. It consists of just two submodules: dram-id memory and modulo k adder. The **dram-id memory** stores for each flow the dram-id that will be delivered at the next request. The **modulo k adder** calculates the next dram-id according to the round robin scheme, i. e., it always adds 1 to the current dram-id.

Concluding, the dispatcher module is quite simple and can be operated at high frequencies.

4.4.4 Dynamic Memory Manager Module

This Section introduces the Dynamic Memory Manager (DMM) module utilized in the tail part. The Section is structured into four paragraphs. The first introduces the tasks of the DMM module. The second points out how the short-cut functionality influences the DMM module

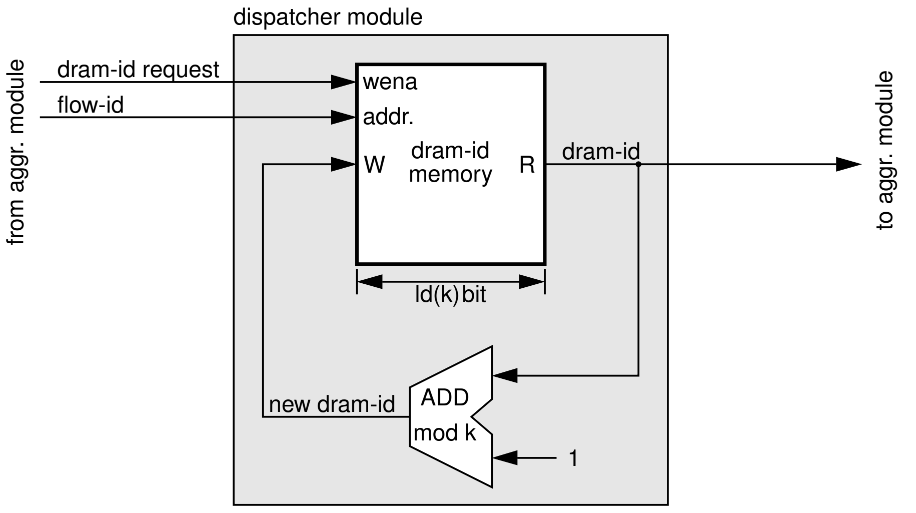


Figure 4.7: Dispatcher module

implementation. The third discusses implementation consideration and the fourth presents its block diagram.

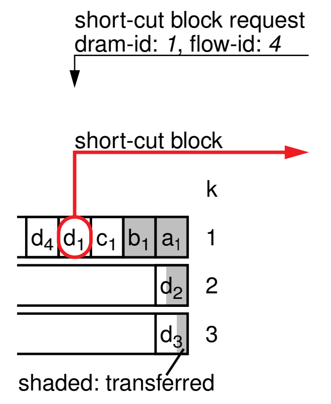
Tasks

According to Section 3.4.1 the DRAM queues have to share tail buffer dynamically in order to reduce the tail buffer's size. The *DMM* module implements the functionality required therefore. It manages a set of queues and dynamically allocates and de-allocates block wise tail buffer space for them. Dynamic memory management is a state-of-the-art functionality and was introduced in Section 2.2.3. However, the DMM module has to provide two additional functionalities. Firstly, it has to operate deterministically, i. e., each clock cycle it has to be able to enqueue and dequeue one block. Secondly, it has to support short-cutting, i. e., allow searching for blocks and short-cutting blocks.

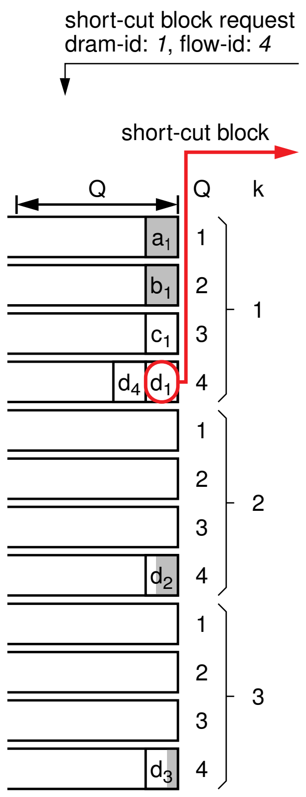
Influence of Short-Cut Functionality

The short-cut functionality has three main impacts on the DMM module implementation: increased number of queues, state information provisioning to short-cut module, and short-cutting of blocks. The following discusses these in detail.

Increased number of queues – According to Chapter 3 the DMM module has to manage k DRAM queues. A DRAM queue holds blocks of all flows. An arriving short-cut request always addresses the headmost block of a flow in an explicit DRAM queue. Consequently, this will require in many cases non-FIFO access to the DRAM queues. Figure 4.8(a) gives an example. The duration of non-FIFO dequeue operations is variable and can be very long. The required amount of time depends on the position of the corresponding block in the DRAM queue. However, to provide a deterministic throughput dequeue operations have to finish in constant time.



(a) k queues (one per DRAM queue)



(b) Qk queues (Q per DRAM queue)

Figure 4.8: Variants for DRAM queue organization in the tail buffer; Example assumes $k = 3$ DRAMs and $Q = 4$ flows named a to d; Both Subfigures show the tail buffer state after the sequential arrival of the blocks a_1, b_1, c_1, d_1, d_2, d_3.

We achieve a constant dequeue time by eliminating non-FIFO accesses. This can be done efficiently by managing Q FIFO queues per DRAM queue[4], where each FIFO queue contains only blocks of a single flow. FIFO access is sufficient, since a short-cut request addresses always the headmost block of a flow. Consequently, the DMM module manages a total of Qk queues. Figure 4.8(b) gives an example for this new DRAM queue organization. Consider that the block requested via short-cut is now the headmost block of the corresponding queue.

This new DRAM queue organization has two impacts on the implementation of the DMM module. Firstly, the queue table holding the queue descriptors (cf. Section 2.2.3) requires Qk instead of just k entries. Secondly, the sequence information between blocks of different flows in a DRAM queue is not available any more. However, due to presence of the short-cut functionality, this is also not required, i. e., the tail transferor can remove blocks from a DRAM queue in any order as long as it is FIFO per flow.

State information provisioning to short-cut module – For short-cut processing the short-cut module requires state information from each module containing blocks, i. e., information about the contained blocks. The DMM module provides two types of state information to the short-cut module. Firstly, it provides dram-id and flow-id of the blocks it currently writes to or reads from tail buffer. Secondly, it provides the basic queue state information of all Qk queues managed in the tail buffer, i. e., empty/non-empty.

Short-cutting of blocks – There are two principle variants to short-cut a block from the DMM module which is currently written to the tail buffer. Firstly, the short-cut module lets the write operation finish and triggers the tail-transferor some clock cycles later to request the block. Secondly, the DMM module aborts the write operation and delivers the block immediately. We chose the second variant for the prototype as it reduces the short-cut read latency, i. e., the time between the arrival of a short-cut request at the tail part and delivery of the corresponding block to the head part. Further, direct processing avoids accumulation of unprocessed short-cut requests and therewith the DMM module needs to provide only basic queue state information to the short-cut module, i. e., empty/non-empty. Section 4.4.5 introduces short-cut functionality in more detail.

Implementation Considerations

The DMM module provides beside state-of-the-art dynamic memory management two additional functionalities: deterministic operation and short-cutting. This paragraph presents implementation considerations related to these additional functionalities.

Deterministic operation – This means the DMM module has to be able to enqueue and dequeue one block each clock cycle. Consequently, the tail buffer has to process one read and one write access per clock cycle. Utilizing standard dual-port memory for the tail buffer satisfies this requirement. However, to enqueue or dequeue a block up to two accesses to the queue table memory are necessary. This leads to a maximum of four accesses

[4]Garcia *et al.* proposed this queue organization in [84]. However, they use it to reorder blocks that arrive out of order.

to the queue table memory per clock cycle what requires operating it at twice the system clock frequency.

Queue state information provisioning for short-cut module and tail transferor module – Both, short-cut module and tail transferor module query the basic state information of the FIFO queues from the DMM module, i. e., empty/non-empty. One bit per FIFO queue is sufficient to store this information. Two reasonable possibilities exist to hold this state information in the DMM module: utilize a Qk bit wide register or a memory (on-chip SRAM) with a size of Qk bit.

Using a register has the pros that a query can be answered during the same clock cycle and that an arbitrary number of modules can read the register simultaneously. However, the con is that each read port requires an up to Qk-times multiplexer, what may limit operation frequency for large Qk.

Using a memory has the pro that it has a very low resource requirement and can operate even for large Qk at high frequencies. However, its con is that on-chip memories have an access latency of typically one clock cycle. This requires pipelining the short-cut request processing.

For the prototype we use a register to hold basic queue state information to simplify short-cut processing. However, the lesson learned is that using a memory (on-chip SRAM) is more reasonable, since pipelining short-cut request processing increases implementation complexity only marginally.

Queue state information provisioning for DMM module itself – For each enqueue and dequeue operation the DMM module has to know the corresponding queue's state in following detail: empty, one element, or more than one element.

To obtain this state information enqueue and dequeue logic can perform a read access to the queue-table memory. From this they can derive the required information, e. g., when head and tail pointer are equal and the queue is valid (non-empty) the queue contains one element. Further, to cope with special cases, like simultaneous enqueue and dequeue to/from a queue currently containing one element, enqueue logic and dequeue logic have to interact.

We chose an alternative solution and use a *counter module* which stores the exact number of elements of each queue. This simplifies the DMM module implementation in two ways. Firstly, it reduces the number of access to the part of the queue table memory that holds the tail pointers. This allows keeping clock frequency identical to the system clock frequency for this part. Secondly, it decouples enqueue and dequeue logic what leads to a more modular implementation.

However, from the current point of view (after prototype implementation) this did not reduce but shifted complexity to a separate module.

Block Diagram

Figure 4.9 depicts the block diagram of the DMM module. It consists of two types of sub-modules: read/write pipeline and memories to hold state information.

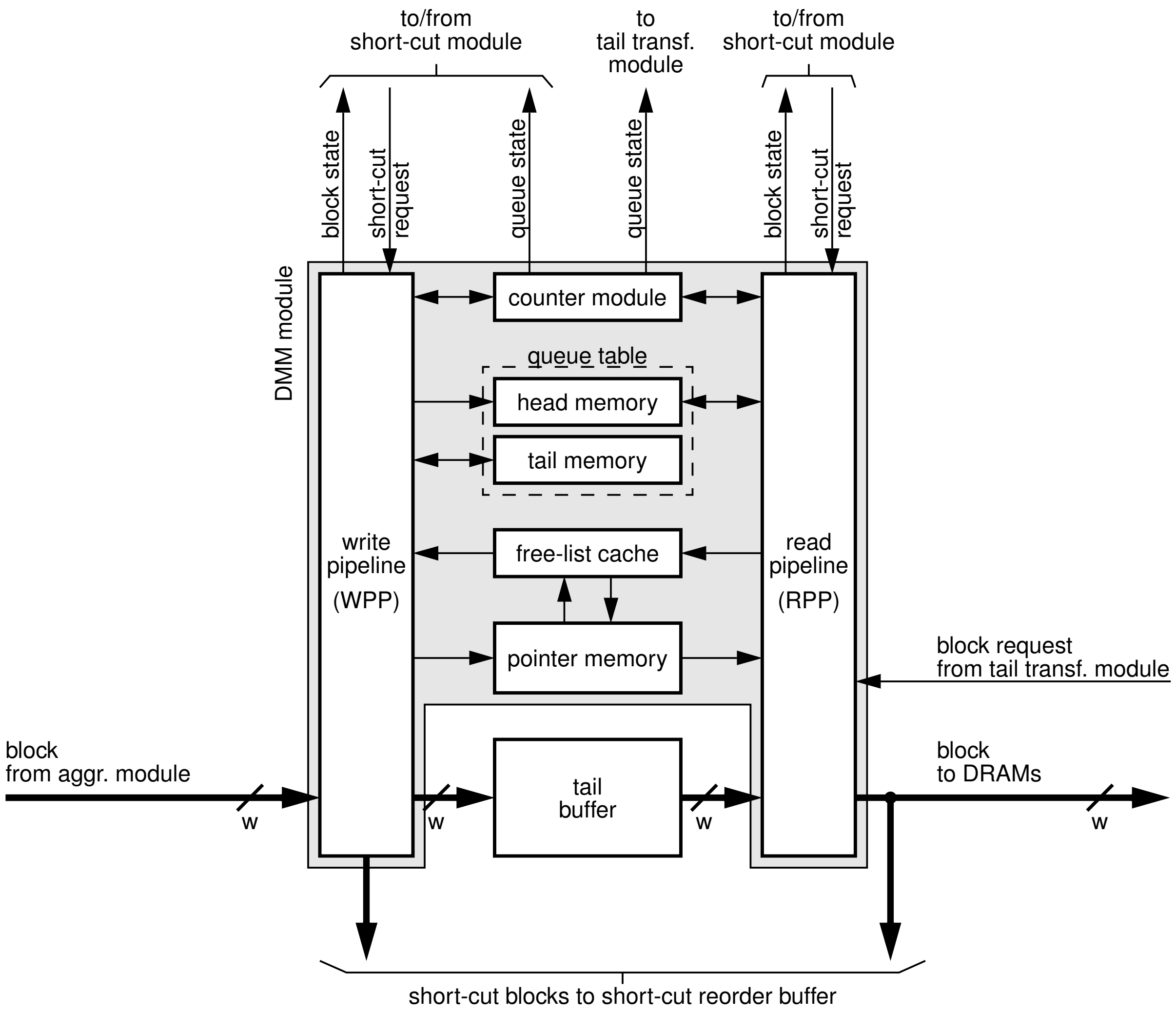


Figure 4.9: Dynamic memory manager module; counter module and head memory are clocked with twice the system clock frequency while all other modules are clocked with system clock frequency

The **Read Pipeline** (RPP) and the **Write Pipeline** (WPP) control the accesses to the memories to perform a block-write (enqueue) or a block-read (dequeue), respectively. The memory accesses they perform are equivalent to those introduced in Section 2.2.3. Using a pipeline enables accepting and delivering one block each clock cycle.

The **counter module** stores for each queue the number of contained blocks. To store this information the corresponding memory has a width of $ld(Q)$ bit as a DRAM queue contains maximally Q blocks. For each increment and decrement the counter module has to read, modify, and write the value back to memory. Consequently, to support simultaneous accesses from WPP and RPP this memory uses a clock with twice the system clock frequency. Further, the counter module maintains a Qk bit wide register containing the basic queue state information (empty/non-empty) of each queue. It provides this information to the short-cut module and tail transferor module.

Due to the presence of the counter module WPP and RPP need selective access to head and tail pointers. To benefit from this, we implement the **queue table** with two separate memories: the **head memory** storing the queues' head pointer and the **tail memory** the queues' tail pointer. Therewith, we could reduce the clock frequency of the tail memory to the system clock frequency.

Finally, the DMM module contains a **free-list cache**. Every clock cycle it can deliver one free element to the WPP and receive one from the RPP. Its presence reduces the number of accesses to the **pointer memory** from four to two per clock cycle, i. e., from $2{\times}R$ and $2{\times}W$ to $1{\times}R$ and $1{\times}W$. The necessary cache size is just a single free element due to the following two observations. Firstly, the cache fill level does not change when WPP fetches and the RPP returns simultaneously a free element. Secondly, when a pipeline does not access the free-list cache it also does not access the pointer memory. The free-list cache uses this free pointer memory bandwidth to store/retrieve free list elements, i. e., it keeps its fill level constant.

Table 4.3 lists the pointer memory and free-list cache accesses for all possible operating conditions. As an example, the first row of the table shows the operating condition where the WPP enqueues a block (store) and the RPP performs no operation (–). Therefore the WPP fetches a free element from the free-list cache (fetch) and writes to the pointer memory to enqueue the new block (W_{queue}). Simultaneously, the free-list cache reads the pointer memory to retrieve a new free element ($R_{freelist}$).

operating condition	**WPP**	**RPP**	**free-list cache**		**pointer memory**	
1	store	–	fetch	–	$R_{freelist}$	W_{queue}
2	–	retrieve	–	return	R_{queue}	$W_{freelist}$
3	store	retrieve	fetch	return	R_{queue}	W_{queue}
4	–	–	–	–	–	–

Table 4.3: Pointer memory and free-list cache accesses at any possible operating condition; '–' stands for *no operation*

Summarizing, the DMM module is complex and strongly influenced by the short-cut functionality and the required deterministic bandwidth. Their largest influence is that some memories in the DMM module have to operate with twice the system clock frequency. However, this is not limiting as the corresponding memories are very small compared to the tail or head buffer size and small memories can run at higher frequency than large ones.

4.4.5 Short-Cut Module

This Section introduces the short-cut module. It is structured into two paragraphs. The first introduces the short-cut module's tasks while the second presents its block diagram.

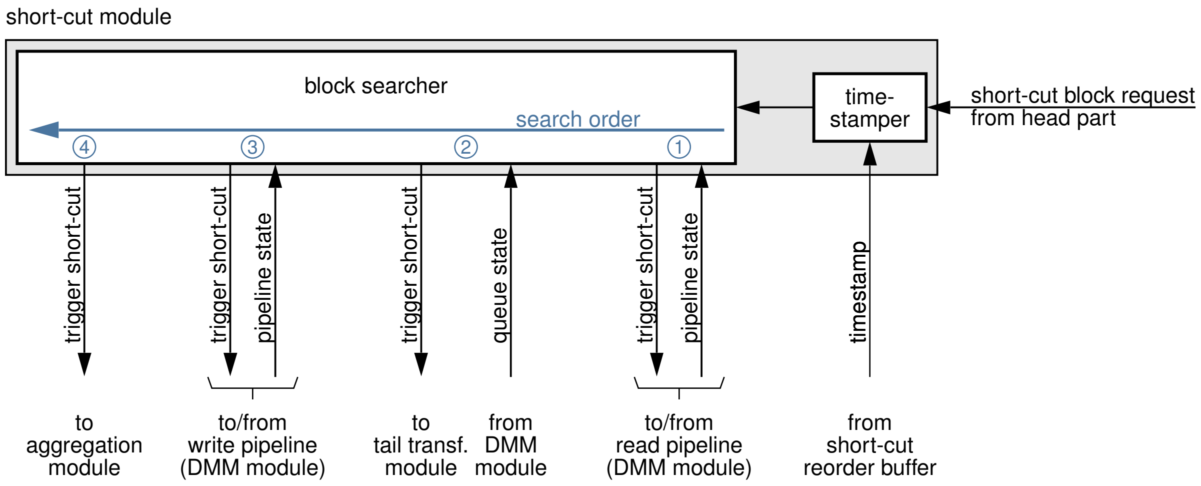


Figure 4.10: Short-cut module

Tasks

The global task of the short-cut module is processing incoming short-cut block requests from the head part. This global task can be subdivided into three tasks, which are described in the following.

Search tail part for addressed block – An arriving short-cut block request always addresses the headmost block of a flow in an explicit DRAM queue, e. g., headmost block of flow 5 in DRAM queue 1. The short-cut module searches the tail part to find the block's location.

Trigger short-cutting of block – Knowing the block's location the short-cut module triggers the corresponding module to short-cut the block.

Timestamp short-cut block request – Retrieval time of a block depends on its location and is either constant or varies within a bounded number of clock cycles. For example, the write pipeline in the DMM module can short-cut a block immediately while retrieval of a block from the tail buffer varies based on the tail transferor's state. To be able to reorder the short-cut blocks, the short-cut module timestamps each incoming short-cut block request. Later on, the short-cut reorder module uses this information for reordering.

Block Diagram

Figure 4.10 depicts the block diagram of the short-cut module. It consists of two sub-modules: *timestamper* and *block searcher*.

The **timestamper** is a simple module that tags each short-cut block request with a timestamp. The short-cut reorder buffer therefore provides a unique time stamp at each clock cycle.

The **block searcher** determines the location of the addressed block in the tail part. Four possible locations exist: DMM module's read pipeline, tail buffer, DMM module's write pipeline, and aggregation module. The search order is from read pipeline to aggregation module. The block searcher receives state information about contained blocks only from the first three locations. When these do not contain the block, it has to be in the aggregation module.

Knowing the location the block searcher sends a trigger to the corresponding module (cf. Figure 4.10). The trigger requests the forwarding of the block to the short-cut reorder buffer. When the block resides in the tail buffer the block searcher sends the trigger to the tail transferor module, which is responsible for requesting blocks from tail buffer.

To reduce prototype implementation complexity the block searcher operates combinatorial. This limits system frequency of the prototype for large Qk as the DMM module provides the queue state information as a Qk wide vector. However, pipelining of the block searcher would remove this limitation and requires just small additional implementation effort. This is a lesson learned. To additionally minimize resource usage, Section 4.4.4 presents an alternative solution for queue state information provisioning.

Summarizing, the short-cut module is a crucial module for short-cut processing. It supports pipelining and therewith scales towards large Qk.

4.4.6 Short-Cut Reorder Buffer

This Section introduces the short-cut reorder buffer and is structured into three paragraphs. The first introduces its tasks. The second discusses implementation considerations, while the third presents its implementation.

Tasks

The individual modules of the tail part deliver short-cut blocks with different and variable latencies. The tail transferor module even processes short-cut requests out of order (cf. Section 4.4.7).

To provide a simple short-cut interface with constant read latency the short-cut reorder buffer performs two tasks. Firstly, it reorders the short-cut blocks according to the order of the requests. Secondly, it delays the individual blocks to achieve the constant read latency.

Implementation Consideration

The short-cut reorder buffer can receive blocks from three different modules: the aggregation module, the DMM module's write pipeline, and the DMM module's read pipeline (cf. Figure 4.3, page 115). Consider that the read pipeline contains only block requests and that a block from the tail buffer is only available in the last stage of the read pipeline.

Due to the different short-cut latencies of the modules, the short-cut reorder buffer has to accept up to three blocks simultaneously, i. e., during one clock cycle. However, in average at most

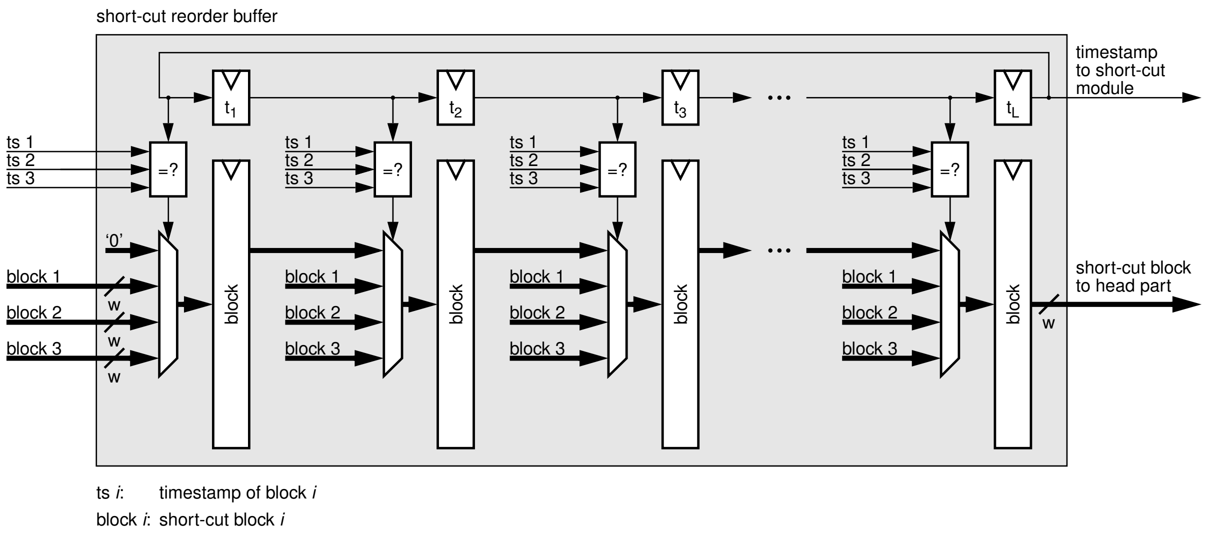

short-cut reorder buffer
timestamp to short-cut module
ts 1
ts 2
ts 3
=?
'0'
block 1
block 2
block 3
w
block
t1
t2
t3
tL
short-cut block to head part
ts i: timestamp of block i
block i: short-cut block i

Figure 4.11: Short-cut reorder buffer

one block per clock cycle arrives at the reorder buffer, since the head part sends at most one short-cut block request per clock cycle.

A simple way to implement a buffer with this amount of write ports is to use registers. This is the approach taken for the prototype, since the buffer has to hold a few blocks only. A more resource efficient solution is using a memory for buffering and a set of registers for absorbing simultaneous writes.

A short-cut block arrives at the reorder buffer earliest l_{min} and latest l_{max} clock cycles after the arrival of the corresponding short-cut block request. To achieve a constant read latency of l_{max} the reorder buffer has to delay a block at most $L = l_{\text{max}} - l_{\text{min}}$ clock cycles. This corresponds to the buffer size in blocks. In this prototype l_{min} and l_{max} have the following values.

$l_{\text{min}} = 0$ (write pipeline forwards a short-cut block immediately)
$l_{\text{max}} = k+2$ (tail transferor's + read pipeline's maximal short-cut latency)

Block Diagram

Figure 4.11 shows the block diagram of the short-cut reorder buffer. Its input interface connects to the three modules that can deliver short-cut blocks. It receives with each short-cut block also the timestamp of the corresponding short-cut block request. On its output interface it delivers the reordered short-cut blocks to the head part.

The implementation consists of $L = k+2$ identical pipeline stages. Each can store one block and one timestamp. Upon startup, each stage is initialized with a unique timestamp. Each clock cycle timestamp and block move to the next stage. Thereby, the input timestamp of the first stage is the output timestamp of the last stage. The reorder buffer provides this input timestamp also to the short-cut module.

Upon arrival of one or more blocks at the input interfaces, each pipeline stage compares its time stamp to that of the arriving blocks. When there is a match, the stage stores the corresponding block. Collisions, i. e., overwriting of a valid block, cannot occur, because a unique timestamp is assigned to each short-cut block request.

Summarizing, the short-cut reorder buffer is a simple and highly pipelined module. Consequently, it does not limit design frequency.

4.4.7 Tail Transferor Module

This Section introduces the *tail transferor module* and is structured into three paragraphs. The first introduces its tasks. The second discusses implementation issues, while the last presents its block diagram.

Tasks

The *tail transferor module* hast two tasks. Firstly, it triggers transmission of blocks from DRAM queues in the tail buffer to the DRAMs according to the *tail-MMA* (cf. Section 3.3.3, page 75). Explicitly, it has to periodically check the state of each DRAM queue in the tail buffer, i. e., every k time slots. Whenever it finds a DRAM queue non-empty, it has to generate a block request for the DMM module to retrieve a block from the corresponding DRAM queue. Secondly, it processes short-cut block requests that have to be served from tail buffer. Therefore, it also generates block requests for the DMM module. These block requests contain a tag, which lets the retrieved blocks follow the short-cut path.

Implementation Considerations

To suffice the *tail-MMA*, the tail transferor module has to check the DRAM queue state and generate block requests for the DRAM queues in strict round robin manner, i. e., $0, 1, \ldots$,k-1$, 0, \ldots$. In full load situation it generates one block request per system clock cycle. Consequently, an immediate processing of an arriving short-cut block request is not possible, as it would violate the round robin sequence.

A possible solution for this is increasing the system clock frequency to have enough free cycles for processing short-cut block requests. However, this puts the feasibility of the whole system at a risk.

We solve the problem by using a buffer that stores arriving short-cut block requests. When the tail transferor module is about to generate a block request for DRAM queue i it also checks this buffer. When there is a short-cut block request for DRAM queue i it forwards this to the DMM module. This means, each short-cut block request leaves the tail transferor module after at most k clock cycles. Further, the head part sends at most every k clock cycles a short-cut block request addressing the same DRAM queue. Consequently, the buffer has to hold at most one short-cut block request per DRAM queue and is therewith negligible in size.

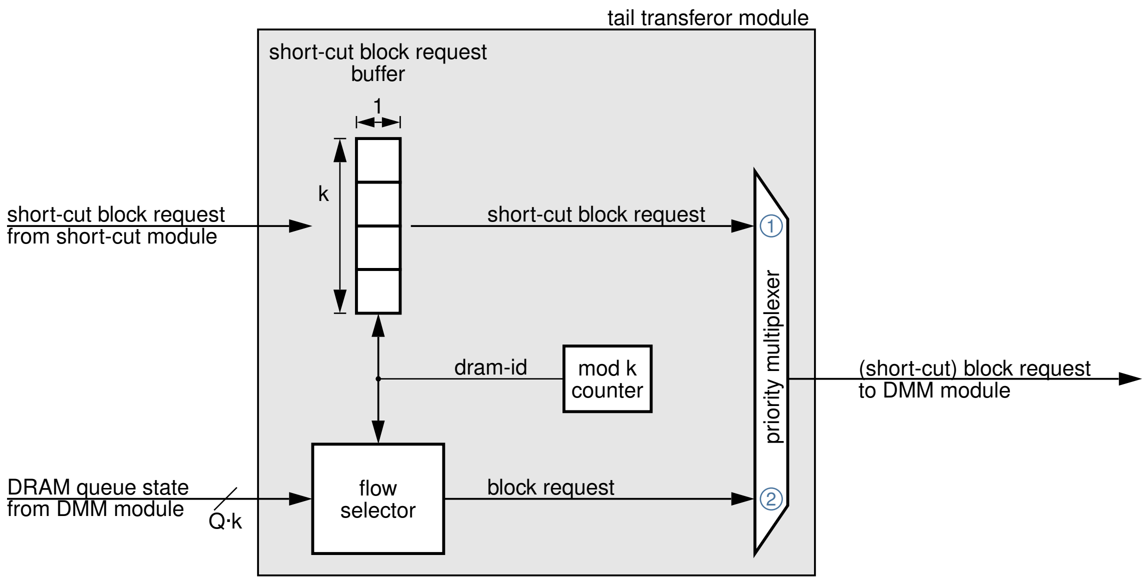

tail transferor module
short-cut block request buffer
1
k
short-cut block request from short-cut module
short-cut block request
dram-id
mod k counter
priority multiplexer
(short-cut) block request to DMM module
DRAM queue state from DMM module
Q·k
flow selector
block request
①
②

Figure 4.12: Tail transferor module

Block Diagram

Figure 4.12 shows the block diagram of the tail transferor module. It consists of four sub-modules.

The **mod k counter** assures that the DRAM queues are processed in round robin sequence. It continuously counts from 0 to $k-1$ and increments its value each clock cycle by one. Its output value serves as *dram-id* and determines the DRAM queue for which the tail transferor module generates the next block request. When the DRAM queue addressed by *dram-id* is empty, it generates no block request in this clock cycle.

Like introduced in Section 4.4.4 the DMM module implements each DRAM queue as a set of Q FIFO queues, i. e., one per flow. The **flow selector** receives the DRAM queues' state information from the DMM module and the *dram-id* from the mod k counter. Based on this, it selects from the DRAM queue addressed by *dram-id* a flow, which's FIFO queue contains a block. Knowing *flow-id* and *dram-id* the flow selector finally generates a block request and delivers it to the priority multiplexer. The flow selection task is pipelineable and therewith it does not limit system clock frequency. The prototype contains a flow selector with one pipeline register. If pipelined the flow selector requires as input a future *dram-id*.

The **short-cut block request buffer** stores the requests arriving from the short-cut module. Since the buffer is very small we implement it using registers.However, an implementation with on-chip memory is also feasible. Each clock cycle the buffer checks if the short-cut block request addressed by the *dram-id* is valid. If valid, it delivers it to the priority multiplexer.

Finally, the **priority multiplexer** chooses the block request with the higher priority and forwards it to the DMM module. Thereby, it always prioritizes a short-cut block request over a block request. This means, in the case when both block requests are valid, it forwards the short-cut block request and discards the block request.

Summarizing, the tail transferor module is the central control unit in the tail part, responsible for block retrieval from the tail buffer. The proposed implementation is capable to run at high frequencies as the sole combinatorially complex task (flow selection) is pipelineable.

4.5 Head Part

This Section presents the block diagram of the SPHSD's head part. Therefore, it introduces the functional requirements of head part and shows the resulting architecture fulfilling these. Finally, it describes the functionality of the individual modules.

In the considered input line card scenario the head part has six main functions. Table 4.4 lists these. Like the tail part, the head part implementation has to provide beside its functions a deterministic bandwidth. The functions can be directly mapped to one or more modules of the head part. Table 4.4 shows this mapping.

	function	**head part module implementing it**
1.	generation of block requests for DRAMs based on arriving cell requests	$\rightarrow$ requester module
2.	buffering block requests	$\rightarrow$ request buffer
3.	triggering block request transfers to DRAM and tail part	$\rightarrow$ head transferor module
4.	determining blocks' location and accordingly forwarding block requests to DRAM **or** tail part	$\rightarrow$ block location checker
5.	provisioning of cells to cell validator with constant read latency	$\rightarrow$ head buffer $\rightarrow$ read latency FIFO
6.	provisioning *last block request of flow i left* information to SIM	$\rightarrow$ sequence number generator $\rightarrow$ last block request buffer $\rightarrow$ block location checker

Table 4.4: Functions of the head part and their mapping to modules

Figure 4.13 shows the block diagram of the head part. In the figure from right to left the **head part** receives cell requests from the scheduler. Consider that in this input buffer scenario each cell request leads to one block request and one block. Upon arrival of a valid cell request the **sequence number generator** generates a sequence number. Based on the same cell request, the **requester module** generates a block request. Therefore, it determines the DRAM storing the block, i. e., the *dram-id*. Then it tags the block request with the sequence number and forwards it to request buffer and read latency FIFO.

The sequence number uniquely identifies a block request in the head part. Therefore the sequence number needs to have a width of only $ld(Q \cdot k)$, since never more than $Q \cdot k$ block requests reside simultaneously in the head part.

Simultaneously to block request generation the **Last Block Request Buffer** (LBRB) stores this sequence number. The LBRB stores the sequence number of the last block request for each flow. We implement it as a memory with $Q \times ld(Q \cdot k)$ bit in size.

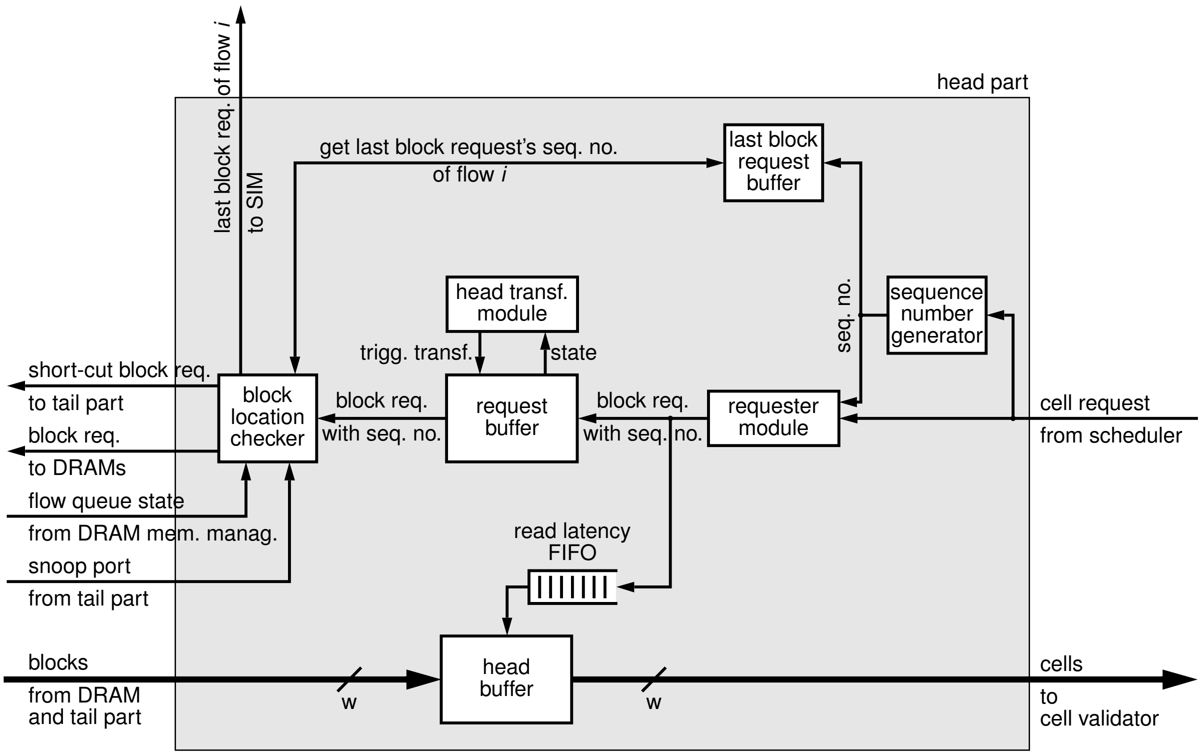


Figure 4.13: Head part

The **request buffer** buffers the block requests including the sequence number and therefore maintains *k* request queues. Further, it provides state information of the individual request queues to the **head transferor module**. The head transferor module checks the state of each request queue periodically, i. e., evey *k* time slots. Whenever it finds a request queue non-empty, it triggers the forwarding of the queue's headmost block to the block location checker.

The **block location checker** has two tasks. Firstly, it checks for each received block request if the corresponding block resides in DRAM or in the tail part. Therefore, it requests the flow queue state information of the corresponding DRAM from the memory manager maintaining the flow queues in the DRAMs. When the flow queue state of the corresponding DRAM is *non-empty* it forwards the block request to the DRAM. When the flow queue state of the corresponding DRAM is *empty* the block still resides in the tail part. Then it forwards the block request to the tail part. These block requests are called short-cut block requests. Since accessing a DRAMs flow queue state information may take some clock cycles, the received state information may be outdated. Therefore, the block location checker additionally snoops the block transfers from tail part to DRAMs. So it can detect, if an addressed block moved from tail part to DRAM during checking.

Secondly, the **block location checker** informs the SIM if the block request leaving the head part is the *last block request of flow i* or not. *i* is the flow-id of the leaving block request. Therefore it requests the last block's sequence number of flow *i* from LBRB and compares it to the sequence number of the leaving block request. The author implements the block location checker as a pipeline.

After a constant read latency, the blocks from DRAMs and tail part arrive at the head part. The **head buffer** buffers the blocks and therefore maintains *k* DRAM queues. Consider that this block arrival is out of order compared to the arrival of the cell requests.

The **read latency FIFO** is responsible to achieve a constant read latency and therewith in-order delivery. Therefore, it buffers each block request received from the requester module for the time of the read latency (cf. Section 3.4.3). After this time, it forwards the block request to the **head buffer**. The head buffer now retrieves the corresponding block, converts it to a cell by removing the dram-id from its meta-information, and forwards it to the *cell validator*.

Implementation of the head part turned out to be less complex compared to the tail part. This has three main reasons. Firstly, the considered input buffer scenario requires no packet reassembly functionality. Secondly, the MiRBD head transferor algorithm in combination with delivery of full blocks (no reassembly) allows to use static memory allocation in the head buffer without increasing its size. Thirdly, the head part just generates short-cut block requests, while the tail part performs the complex processing of the short-cut block requests.

Summarizing, the head part's implementation is complex but at the same time very modular with narrow interfaces. Since all its modules can be implemented pipelined, its operating frequency is only limited by the head buffer size.

4.6 Validation

This Section presents the validation of the implemented architecture and covers the main aspects functionality, resource requirement, and supported line rate. Section 4.6.1 introduces the functional tests performed to show that the prototype is fully functional. Section 4.6.2 introduces resource requirement and throughput of the prototype implementation. Section 4.6.3 summarizes the validation results.

4.6.1 Functional Tests

The introduced prototype has been tested for correct functionality in different parameterizations and under different load situations, including full load situation. A functional test under full load is at the same time also a throughput test, since the proposed SPHSD architecture provides a deterministic bandwidth and a constant read latency. Section 4.3.1 starting on page 107 already introduced the overall prototype structure used for functional testing.

For the sake of completeness two further properties of the prototype are introduced here. Firstly, the contained scheduler supports two scheduling strategies. Both request cells from the individual flows in a round robin manner. The difference is, that the one requests all the data currently buffered from a flow, while the other requests each time just a single cell from a flow.

Secondly, we set the read latency of the DRAM model to be equal to the constant read latency of a short-cut request. Therewith, the merger in Figure 4.2 on page 108 is a simple multiplexer.

In a real system an additional small FIFO buffer may be necessary delaying the blocks of the faster module.

Two types of functional tests were performed: clock cycle accurate simulation and operation on FPGA. To achieve many different configurations the system parameters were varied, i. e., bus width w, degree of parallelism k, and number flows queues Q.

Test via clock cycle accurate simulation – The author used the simulations software *Modelsim* [98] from Mentor Graphics to simulate individual modules as well as the complete prototype. Clock cycle accurate simulation is computationally intensive, i. e., simulation of milliseconds takes several minutes of real time on a standard PC. Consequently, to test the correct operation of SPHSD implementation with a huge number of incoming packets, tests on real hardware are mandatory, e. g., on an FPGA.

Test on an FPGA – The tests on FPGA were done on an UHP-2 board like shown in Figure 4.1 on page 106. The test run for each configuration of the SPHSD was in the range of minutes to hours. This means in each test run many gigabytes to terabytes of data in form of many billion blocks passed the SPHSD. Each test run was performed with different parameterizations of the random number generators in the packet generator to cover many different packet arrival patterns. Tests included low and full load situations.

A test run passes, when the requested cells arrive after a constant read latency at the cell validator (cf. Figure 4.2 on page 108) and contain the correct data. The prototype passed all test runs performed with the final version of the implementation. Concluding, the SPHSD architecture is fully operational and its implementation in hardware is feasible.

4.6.2 Hardware Resource Requirement and Supported Line Rate

This Section presents the hardware resource requirement and supported line rate of the prototype implementation. Therefore, the first paragraph introduces the conditions, like used tools, settings, and target FPGA devices. The second paragraph introduces the metrics to interpret the result, while the third introduces the system parameters and assumptions. The last paragraph presents the explicit hardware resource requirements for different configurations of the prototype along with the correspondingly supported line rate.

Conditions

As target FPGA for analysis the author uses an Altera Stratix II FPGA of the type EP2S 60 F1020 C3 [96], which is available on the UHP-2 board used for functional testing. For prototype configurations, which do not fit on this FPGA the author uses an Altera Stratix II FPGA of the type EP2S 130 F1020 C3 [96] with similar properties but higher logic block count.

Synthesis of the prototype's VHDL description was performed with the software *Precision RTL Synthesis*, version 2011a [99] from Mentor Graphics. The tool was set to use *register retiming* as this improves circuit performance [106]. Further it was set to preserve the hierarchical boundaries of the head part and the tail part. This enables reporting the exact resource requirements utilized by the head and the tail part as *Precision* now performs optimizations including register retiming only inside and outside of these two modules.

Place & Route was performed with *Quartus II*, version 11.0 [100] from Altera. Quartus II delivers information about the resource requirement as well as the maximal achieved clock frequency. Quartus II was used with its default settings.

To achieve results reflecting the limits of the SPHSD implementation the author replaced the following modules with dummy modules: DRAM model, SIM, Scheduler. The dummy modules feature the equal registered interfaces as the original modules. As *Precision* was set to preserve the hierarchical boundaries of the head and the tail part the results regarding the SPHSD are realistic despite using dummy modules.

Metrics

Stratix II devices consist of matrix of *Adaptive Logic Modules* (ALM) [96]. An ALM mainly consists of a set of look-up table (LUT) based resources and two Flip-Flops (FF). The LUT resources of an ALM can be divided between two so called *Adaptive LUTs* (ALUT). Consequently, an ALM can implement two *ALUT/FF pairs*, each consisting of a combinational logic portion and a FF.

Therewith, an ALUT/FF pair can implement a combinational function with subsequent FF, just a combinational function (without FF), or just a FF. The used EP2S60 device contains 24 176 ALMs corresponding to 48 352 ALUT/FF pairs, while the EP2S130 device contains 53 016 ALMs corresponding to 106 032 ALUT/FF pairs.

This thesis presents the resource usage in used ALUT/FF pairs like provided by the Quartus II software. The values represent the number of uses ALUT/FF pairs after place & route.

An FPGA contains a large number of small SRAM memories, called *RAM-blocks*, distributed over the whole FPGA. The used EP2S60 device contains a total of 2 485 kbit of memory, provided by 329 M512-RAM-blocks (each 576 bit), 255 M4K-RAM-blocks (each 4.5 kbit), and 2 M-RAM-blocks (each 576 kbit). The used EP2S130 device contains a total of 6 590 kbit of memory, provided by 699 M512-RAM-blocks, 609 M4K-RAM-blocks, and 6 M-RAM-blocks.

The memory usage given in this thesis corresponds to the number of bits required by a module, i. e., the amount given in the module's VHDL implementation. The actually occupied memory resources on the FPGA are always larger or equal than this. This has two reasons. Firstly, the Place & Route tool Quartus II always instantiates memories with word numbers being a power of two. This leads in worst case to an overdimensioning by nearly 100,%, e. g., use of 1024 instead of 513 words. The designer can avoid this overdimensioning by manually decomposing the memory into memories with word numbers being a power of two. Such a change influences timing of the design. Secondly, due to the fixed sizes of the RAM-block types there is often fragmentation when they are used to implement a memory with custom width and depth.

Given clock frequencies represent the maximal achievable clock frequency. Quartus II derives it from the critical path in the design, which corresponds to the longest FF to FF delay.

System Parameters and Assumptions

Resource requirements and throughput results are presented for several parameterizations of the prototype. The main configurable parameters of the prototype implementation are the bus width w (equal to the block size b), degree of parallelism k, and the number of flows Q. These can be set before synthesis.

Additionally, assumptions are made about the DRAM random access time T and the used network technology. Consider that these two assumptions only influence the clock frequency of the system. For the DRAM random access time we assume a typical value of $T = 50\,\text{ns}$. As network technology we assume an Ethernet based system, i. e., $P_{\text{min}} = 64\,\text{byte}$ and $G_{\text{min}} = 20\,\text{byte}$.

Since the SPHSD system delivers deterministic bandwidth, one can derive all further parameters of the system based on the ones mentioned. With Eq. (3.1) on page 71 the supported line rate R of a configuration can be calculated:

$$w = b = \frac{2TR}{k} \qquad \rightarrow \qquad R = \frac{wk}{2T} \qquad \text{e. g.} \qquad R = \frac{512\,\text{bit} \cdot 4}{2 \cdot 50\,\text{ns}} = 20.48\,\text{Gbps}$$

Considering additionally the used network technology the minimal required system clock frequency can be calculated based on Eq. (4.4) on page 112:

$$f \geq \frac{2R}{w + 1\,\text{byte} + G_{min}} \qquad \text{e. g.} \qquad f \geq \frac{2 \cdot 20.48\,\text{Gbps}}{512\,\text{bit} + 1\,\text{byte} + 20\,\text{byte}} = 60.2\,\text{MHz}$$

Implementation Results

Table 4.5 shows the hardware resource requirements and the supported line rates for different configurations. The table contains three main parts: the *configured* parameters (k, w, Q), the *calculated* values (R and f), and the achieved *system clock frequency and resource requirements*. As the SPHSD provides a deterministic bandwidth, the calculated line rate R corresponds to its throughput.

Recall for reviewing of these results, that the prototype implementation achieves full functionality, but is not optimized for throughput (e. g., high frequency) or low resource usage (e. g., design contains Management-System for configuration an debugging). Nevertheless, the implementation supports a line rate of 10 Gbps easily and fails support of 20 Gbps only marginally (due to an 8 MHz too low frequency).

Independent of the prototype configuration in Table 4.5, the critical path is always a path to the counter module (cf. Figure 4.9 on page 125). The counter module operates at twice the system clock frequency, i. e., signals interfacing with the counter module have to have a path delay that is less or equal the half system clock period.

configuration			calculatation		system clock frequency and hardware resource requirements						
						prototype$^+$	tail part		head part		
k	w [bit]	Q	R [Gbps]	$f_{required}$ [MHz]	f_{max} [MHz]	ALUT/FF [#]	ALUT/FF [#]	memory [kbit]	ALUT/FF [#]	memory [kbit]	Stratix II FPGA
4	128	64	5,1	34,6	50,0	17010 (35 %)	5005	38,9	1782	40,9	EP2S 60 F1020 C3
4	256	64	10,2	48,3	53,0	23168 (48 %)	8669	71,1	2829	73,2	EP2S 60 F1020 C3
4	512	64	20,5	60,2	52,5	36116 (75 %)	16710	135,2	4876	137,4	EP2S 60 F1020 C3
4	1024	64	41,0	68,7	41,5	66066 (62 %)	36000	263,3	9049	265,7	EP2S 130 F1020 C3
8	512	64	41,0	120,5	45,1	47234 (45 %)	22497	206,7	10159	275,4	EP2S 130 F1020 C3
4	512	128	20,5	60,2	42,2	41104 (85 %)	18509	272,1	5007	277,0	EP2S 60 F1020 C3

Table 4.5: Hardware resource requirements and supported line rates of the prototype implementation a different configurations; Values in parenthesis refer to the relative amount of ALUT/FF resources occupied on the FPGA; $^+$ dummy modules used for DRAM model, SIM, and scheudler

According to Table 4.5 the tail part uses twice or more ALUT/FF pairs compared to the head part. This difference origins from the aggregation and short-cut functionality additionally implemented by the tail part. Further, one can see, that the ALUT/FF pairs required for head and tail part increase roughly proportional with the bus width w.

In the following we focus on the memory requirement (SRAM). Table 4.5 shows the total memory utilized by head part and tail part. To the memory of the head part contribute the head buffer, the request buffer, the read latency FIFO, and several other smaller memories (cf. Section 4.5). To the memory of the tail part contribute the tail buffer (stores only full blocks), the aggregation memory, the pointer memory, and several other smaller memories (cf. Section 4.4).

Regarding the memory requirement of head and tail part, one can see, that for $k = 4$ the tail part requires only marginally less memory compared to the head part. This has four main reasons.

- Firstly, with such a small value for k tail buffer size decreases only medium in size, i. e., 37.5 % for $k = 4$.
- Secondly, for small k the aggregation memory (Q blocks) and the tail buffer ($Q\frac{k-1}{2}$ blocks) are similar in size (cf. Section 4.4.2 for formulas), e. g., for $k = 4$ the tail buffer is $1.5Q$ blocks. To support short-cutting the aggregation memory requires a second read port. We implemented this by doubling the aggregation memory, i. e., the total aggregation memory size is $2Q$ blocks.
- Thirdly, as the Place & Route tool Quartus II can only instantiate memories with word numbers being a power of two, the author rounded up all memory word numbers in the tail part to the next power of two in implementation. For example, the topmost configuration in Table 4.5 requires theoretically $Q\frac{k-1}{2}$ blocks = 12 kbit but the author implements it as 16 kbit memory. This is not true for the head part, but there anyway all relevant memories have word numbers being a power of two due to the chosen configurations. To detect tail buffer fill levels that exceed the theoretical bound from Section 3.4.1 the design contains a *fill level checker* module. Expectedly, this never detected an exceeding.
- Fourthly, the dynamic memory management implemented in the tail part introduces an overhead in form of the pointer memory, queue table, etc.

For larger k the tail part implementation utilizes in total significantly less memory than the head part, e. g., cf. configuration with $k = 8$ in Table 4.5. Two facts contribute to this. Firstly, increasing k leads to a decrease in tail buffer size. Secondly, as at a given line rate R the block size decreases with increasing k, the contribution of the aggregation memory is smaller.

4.6.3 Summary and Conclusions

We validated the prototypical SPHSD implementation with respect to functionality, resource requirement, and supported line rate. The functional tests (simulation and operation on an FPGA) showed that the SPHSD architecture is fully operational and that its implementation in hardware is feasible.

The SPHSD's tail and head buffer sizes are conform to the formulas in Chapter 3. In its tail part it requires additional memory capacity to support short-cutting and dynamic memory management. Consider that related architectures (HSD and CFDS, cf. Section 2.5.2) also require dynamic memory management in the tail buffer.

The presented SPHSD implementation supports a line rate of over 10 Gbps supporting $Q = 64$ flow queues despite using a 6 year old FPGA. The possible implementation optimizations described in Section 4.4 paired with a cutting-edge FPGA will enable much more flow queues and significantly higher system frequencies simultaneously. According to Section 4.3.2 an Ethernet based 100 Gbps system requires a minimal system frequency of 148.8 MHz. Concluding, an FPGA implementation of an SPHSD based packet buffer for $R = 100$ Gbps is challenging, but should be feasible.

Kuon *et al.* compare in [107] FPGAs and semi-custom ASICs. Both devices were manufactured with a 90 nm CMOS process technology. Kuon *et al.* found experimentally, that ASICs achieve a 3 to 4 time shorter critical path delay compared to FPGAs, i. e., the same circuitry can run at a 3 to 4 times higher frequency on an ASIC. Therewith, even the current SPHSD implementation could run at approximately $4 \cdot 50\,\text{MHz} = 200\,\text{MHz}$ on an ASIC with the configurations in Table 4.5. With the provided implementation optimizations (cf. Section 4.4) and due to the fact that critical modules can be pipelined a redesign is expected to run at much higher frequencies despite using larger values for the parameters k, w, and Q. Concluding, we estimate a throughput optimized ASIC implementation of an SPHSD based packet buffer to support a line rate of 100 Gbps and far more.

5 Conclusions

5.1 Summary

This thesis introduced a novel hybrid memory architecture – named SPHSD – for high-speed packet buffers that delivers deterministic bandwidth. The novelty of the SPHDS architecture is that it significantly reduces the memory resources compared to related architectures from literature. Memory resources refer to the required capacity and bandwidth of SRAM and DRAM, as well as to the DRAM data bus pin count. The author implemented a packet buffer based on the SPHSD that shows the feasibility of the SPHSD in real hardware at high speeds. The following paragraphs summarize the thesis in larger detail.

To motivate the necessity of new memory architectures for high-speed packet buffers Chapter 2 introduced the main requirements a high-speed packet buffer has to fulfill. On the functional level it has to provide a set of FIFO queues that hold the packet data. On the technological level it has to provide a specific capacity, bandwidth, and access time. Comparison with state-of-the-art memory devices of different memory technologies showed that devices of a single memory technology cannot fulfill all technological requirements simultaneously. For example, for a 100 Gbps line card already 5 DDR3 SDRAM devices in parallel fulfill the capacity requirement but they fail the access time requirement by a factor of 20. The basic architectural concepts to solve the shortcomings are parallelism, interleaving, and aggregation.

A hybrid memory architecture enables to build high-speed packet buffers that provide a deterministic bandwidth. Hybrid refers to the fact that it utilizes memory devices of two different memory technologies: SRAM and DRAM. The architecture combines the strengths of both memory technologies: short access time of SRAM *and* large capacity of DRAM. Chapter 2 introduced the necessary metrics to evaluate hybrid memory architectures. Then it surveyed and evaluated the three types of hybrid memory architectures providing deterministic bandwidth available in literature. These were namely HSD, CFDS, and PHSD (only partially provides a deterministic bandwidth). Each uses a different subset of the aforementioned basic architectural concepts. Finally, Chapter 2 summarized and compared the pros and cons of the surveyed architectures.

In Chapter 3 a novel hybrid memory architecture for high-speed packet buffers that delivers deterministic bandwidth was presented. The main objective was to reduce resource requirements without reducing functionality. The design targets were:

- Small SRAM size
- Minimal DRAM resources
  - Utilization of the total DRAM capacity, i. e., no suffering from fragmentation
  - Minimization of the total required DRAM bandwidth
  - High DRAM data bus utilization (reduces DRAM data bus pin count)

The author combined the best approaches of related architectures and enriched them with own approaches to form a new architecture. Section 3.2 derived the explicit architectural features that are necessary to meet all targets simultaneously. The proposed architecture's name is Semi-Parallel Hybrid SRAM/DRAM system (SPHSD). The name reflects its two main properties. Firstly, it is a hybrid memory architecture. Secondly, it contains a set of k parallel DRAMs (or DRAM banks) but just a singe tail buffer (SRAM) and a single head buffer (SRAM). Packets are aggregated to blocks and only blocks are buffered in SRAM and DRAM.

The two unique features of the SPHSD architecture are:

- DRAM queues dynamically share the tail buffer. This significantly decreases tail buffer size compared to other architectures. The same holds also for the head buffer when the proposed MaRBD head-MMA is used.
- The degree of parallelism (i. e., the value of k) can be *freely* chosen, e. g., it is not limited by the number of banks in a DRAM device. k impacts the architecture in two ways. Firstly, the tail buffer size decreases towards increasing k. Secondly, the total available DRAM banks have to be organized into k groups. As DRAM chips have discrete properties (e. g., data bus pin count, number of banks) k influences the DRAM overdimensioning. The designer can now choose a k that minimizes both, tail buffer size and DRAM overdimensioning. Section 3.5 showed an according dimensioning example.

The author utilizes a tail-MMA and a head-MMA (here called MiRBD) from literature and proposes a new head-MMA (MaRBD). An MMA defines how blocks are distributed to the DRAMs and how blocks are transferred between SRAM and DRAM. Using the tail-MMA and the MiRBD head-MMA the metrics of the new architecture were derived and formally proven: upper bounds for the tail buffer size and head buffer size, as well as the read latency. Using the MaRBD head-MMA the upper bound of the head buffer was derived and validated via software simulation. Table 3.2 on page 95 summarizes the resulting metrics.

Quantitative comparison to the surveyed architectures showed the following improvements. The SPHSD reduces the tail buffer size by 47 % to 53 % depending on the compared architecture. Utilizing the MiRBD head-MMA the head buffer size is roughly equal to that of the surveyed architectures. Explicitly, it increases by 6 % or decreases by 12 % depending on the compared architecture. Utilizing the proposed MaRBD head-MMA the head buffer size decreases by approx. 50 %. Further, both head-MMAs require a lower head buffer bandwidth compared to HSD and CFDS, i. e., $2R$ instead of $3R$, where R is the input line rate of the packet buffer. The PHSD has no head buffer. The read latency of the SPHSD is the same for any head/tail-MMA combination. Its value is equal or slightly lower depending on the compared architecture. For comparison to the CFDS the use of high-speed DDR3 SDRAM was assumed.

To minimize DRAM resources, the SPHSD supports the following four features: it eliminates internal and external fragmentation in DRAM, it requires no DRAM bandwidth speedup, and it supports DRAM bank interleaving. Elimination of fragmentation and the absence of speedup reduces the required DRAM bandwidth to the theoretical minimum of $2R$. Bank interleaving enables to maximize the DRAM data bus utilization what minimizes the number of required DRAM data bus pins. Comparison to the surveyed architectures showed, that only the SPHSD supports all four features simultaneously (cf. Table 3.1 on page 95).

Reduction of memory resource requirements leads to reduction of implementation costs and a better scalability. Beside these, less memory resources inevitably lead to a decrease in power consumption, as a packet consist for the most part from memory.

Chapter 4 presented exemplarily the prototypical implementation of a packet buffer for an input line card utilizing the SPHSD. The implementation targets were to show:

- Functionality of the SPHSD – Implementation in hardware respects all details of an architecture. It is therefore well suited to show that the SPHSD is operational.

- Feasibility of the SPHSD architecture in hardware supporting high line rates – Application of the SPHSD in a router is only possible, when the SPHSD can be implemented in hardware and the implementation supports reasonable line rates.

In a first step, Chapter 4 motivated and introduced the overall prototype structure. The prototype was described in VHDL and an FPGA development board served as platform. In a second step, it presented in detail the implementation of the head and tail part of the SPHSD. In a third step, it evaluated the architecture with respect to the targets, i. e., functionality and feasibility in hardware.

Two types of functional tests were performed: clock cycle accurate simulation and operation on FPGA. Functionality of the prototype was tested with many different parameterizations. A packet generator served as data source and a cell validator served as data sink. All testes passed.

Feasibility of the architecture with its MMAs in hardware was also shown by implementation. The prototype supports a line rate of over 10 Gbps providing $Q = 64$ FIFO queues despite using an over 6 year old FPGA. The objective of the implementation was to achieve full functionality. Optimization for high line rates R and/or a large number of FIFO queues requires a second iteration of the implementation. The proposed implementation optimizations described in Section 4.4 paired with a cutting-edge FPGA will enable significantly higher line rates and much more flow queues simultaneously. Therewith even a line rate of $R = 100$ Gbps can be feasible on a cutting-edge FPGA. The author estimated a new optimized implementation for an ASIC to support a line rate of $R = 100$ Gbps and far more (cf. Section 4.6.3).

Concluding, the SPHSD requires significantly less memory resources compared to related architectures from literature, while providing the same functionality. Further more, it is feasible in hardware at high line rates. This leads to two main advantages compared to related architectures: (i) improved scalability towards higher line rates and more queues and (ii) reduced power consumption.

5.2 Limitations

The SPHSD has limitations concerning scalability towards extreme requirements and read latency.

Scalability – Head and tail buffer size of the SPHSD are proportional to the following parameters: line rate *R*, number of queues *Q*, and random access time *T* of the used DRAM chips. If any of these parameters gets very large the head and tail buffer get very large, too. At some point this makes implementation impractical. For example, core routers exist that require tens of thousand of queues.

Read latency – The constant read latency provided by the SPHSD is proportional to the number of queues *Q* and the DRAM random access time *T*. Consequently, large values of *Q* or *T* lead to a large read latency. It depends on the application whether this is tolerable or not. For example, in case of an input buffer this depends on the switch fabric and the switch fabric scheduler. To reduce or even completely eliminate the read latency the designer can utilize the MDQFP head-MMA in the SPHSD. The MDFQP head-MMA was proposed by Iyer *et al.* in [45] and allows to reduce the read latency by increasing the head buffer size.

5.3 Outlook

Further work could formally prove the upper bound of the head buffer size when utilizing the MaRBD head-MMA. Multicast support could be added to enable the application of the SPHSD also in multicast routers. In [11] Iyer addresses the implementation of multicast in high-speed routers.

Finally, the energy efficiency of the SPHSD could be further increased. A basic idea to achieve this is keeping the tail buffer nearly full even in low load situations. This may increase the use of the potentially chip-internal short-cut path and save the energy intensive data transfer via the external DRAM bus. Due to the presence of the short-cut path this would not increase the read latency. A corresponding tail-MMA could therefore apply the same strategy as the MaRBD head-MMA does. Another possibility is to allow parts of the architecture to be powered off.

Bibliography

[1] Arthur Mutter. A novel hybrid memory architecture with parallel DRAM for fast packet buffers. In *Proceedings of the IEEE Workshop on High Performance Switching and Routing (HPSR)*, pages 44–51, June 2010. `doi:10.1109/HPSR.2010.5580282`.

[2] Simon Hauger, Thomas Wild, Arthur Mutter, Andreas Kirstädter, Kimon Karras, Rainer Ohlendorf, Joachim Scharf, and Frank Feller. Packet processing at 100 Gbps and beyond—challenges and perspectives. In *Proceedings of the 10. ITG Symposium on Photonic Networks*, Leipzig, May 2009.

[3] Arthur Mutter, Martin Köhn, and Matthias Sund. A generic 10 Gbps assembly edge node and testbed for frame switching networks. In *Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom2009)*, April 2009. `doi:10.1109/TRIDENTCOM.2009.4976201`.

[4] Arthur Mutter, Sebastian Gunreben, Wolfram Lautenschläger, and Martin Köhn. A testbed for validation and assessment of frame switching networks. In *Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom2010)*, 2010. `http://www.ikr.uni-stuttgart.de/Content/Publications/Archive/Mu_TridentCom2010_40010.pdf`.

[5] Wolfram Lautenschläger, Arthur Mutter, and Sebastian Gunreben. Frame assembly in packet core networks – overview and experimental results. In *Proceedings of the 10. ITG Symposium on Photonic Networks*, Leipzig, Germany, May 2009. `http://www.ikr.uni-stuttgart.de/Content/Publications/Archive/Mu_FrameAssemblyInPacketCore_36813.pdf`.

[6] S. Hauger, S. Junghans, A. Mutter, and D. Sass. A flexible microprogrammed packet classifier for edge nodes of transport networks. In *Proceedings of the 7. ITG Symposium on Photonic Networks*, Leipzig, 2006. `http://www.ikr.uni-stuttgart.de/Content/Publications/Archive/Hg_UST-IKR-MicroprogrammedClassifier_36486.pdf`.

[7] Michael Lebschi. Entwurf und Realisierung eines parametrisierbaren Speichermanagers in VHDL für einen FPGA-basierten Netzknoten. Studienarbeit, Institut für Kommunikationsnetze und Rechnersysteme, Universität Stuttgart, August 2008.

[8] Philipp Müller. Entwurf und Realisierung einer parametrisierbaren Eingangsstufe für einen hybriden Paketpuffer in VHDL. Studienarbeit, Institut für Kommunikationsnetze und Rechnersysteme, Universität Stuttgart, November 2010.

[9] J. Sommer, S. Gunreben, A. Mifdaoui, F. Feller, M. Köhn, D. Sass, and J. Scharf. Ethernet - A Survey on its Fields of Application. *IEEE Communications Surveys and Tutorials*, pages 263–284, 2010. doi:10.1109/SURV.2010.021110.00086.

[10] Itamar Elhanany and Mounir Hamdi. *High-performance Packet Switching Architectures*. Springer-Verlag London Limited, 2006.

[11] Sundar Iyer. *Load Balancing and Parallelism for the Internet*. PhD thesis, Stanford University, 2008. http://yuba.stanford.edu/~sundaes/Dissertation/sundar_thesis_1sided.pdf.

[12] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. Sizing router buffers. *SIGCOMM Comput. Commun. Rev.*, 34(4):281–292, 2004. doi:10.1145/1030194.1015499.

[13] ISO. International Standard 7498–1: Information technology – Open Systems Interconnection – Basic Reference Model: The basic model, 1994.

[14] Paul J. Kühn. *Lecture Notes: Communication Networks 2*. IKR Institut für Kommunikationsnetze und Rechnersysteme, Pfaffenwaldring 47, 70549 Stuttgart, 2007 edition, 2007.

[15] Deepankar Medhi and Karthikeyan Ramasamy. *Network Routing - Algorithms, Protocols, and Architectures*. Elsevier Inc., 2007.

[16] IEEE Computer Society. 802.1Q: IEEE Standard for Local and Metropolitan Area Networks–Virtual Bridged Local Area Networks, 2005.

[17] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 41:406–424, March 1953.

[18] F.A. Tobagi and T.C. Kwok. The tandem banyan switching fabric: a simple high-performance fast packet switch. In *Proceedings of the Tenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '91)*, pages 1245 –1253 vol.3, apr 1991. doi:10.1109/INFCOM.1991.147647.

[19] N. McKeown, C. Calamvokis, and S.-T. Chuang. A 2.5tb/s lcs switch core. In *Hot Chips*, Aug 2001.

[20] M.G. Hluchyj and M.J. Karol. Queueing in high-performance packet switching. *Selected Areas in Communications, IEEE Journal on*, 6(9):1587–1597, Dec 1988. doi:10.1109/49.12886.

[21] N. McKeown. The iSLIP scheduling algorithm for input-queued switches. *Networking, IEEE/ACM Transactions on*, 7(2):188 –201, apr 1999. doi:10.1109/90.769767.

[22] Cisco Systems Inc. *Cisco CRS-1 Carrier Routing System, Brochure*, June 2011. `http://www.cisco.com/en/US/prod/collateral/routers/ps5763/prod_brochure0900aecd800f8118.pdf`.

[23] Will Eatherton. The push of network processing to the top of the pyramid. Presentation at ANCS 2005, 2005. `http://www.cesr.ncsu.edu/ancs/slides/eathertonKeynote.pdf`.

[24] H. Jonathan Chao and Liu Bin. *High Performance Switches and Routers*. John Wiley & Sons, Inc., August 2006. `doi:10.1002/0470113952`.

[25] Simon Hauger. *Architektur für flexible Paketverarbeitung in Hochgeschwindigkeitskommunikationsnetzen*. PhD thesis, University of Stuttgart, July 2011.

[26] Simon Hauger. A novel architecture for a high-performance network processing unit: Flexibility at multiple levels of abstraction. In *Proceedings of the IEEE International Conference on High Performance Switching and Routing*, June 2009. `doi:10.1109/HPSR.2009.5307421`.

[27] Jorge García, Jesús Corbal, Llorenç Cerdà, and Mateo Valero. Design and implementation of high-performance memory systems for future packet buffers. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*, page 373, Washington, DC, USA, 2003. IEEE Computer Society.

[28] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *SIGCOMM Comput. Commun. Rev.*, 19:1–12, August 1989. `doi:10.1145/75247.75248`.

[29] DDR3 SDRAM standard, JESD79-3D, Sep 2009. `http://www.jedec.org/`.

[30] S. Dharmapurikar, S. Kumar, J. Lockwood, and P. Crowley. Optimizing memory bandwidth of a multi-channel packet buffer. In *Global Telecommunications Conference, 2005. GLOBECOM '05. IEEE*, volume 1, page 6 pp., nov.-2 dec. 2005. `doi:10.1109/GLOCOM.2005.1577359`.

[31] Jahangir Hasan, Satish Chandra, and T. N. Vijaykumar. Efficient use of memory bandwidth to improve network processor throughput. *SIGARCH Comput. Archit. News*, 31(2):300–313, 2003. `doi:10.1145/871656.859653`.

[32] Abhijit K. Choudhury and Ellen L. Hahne. Dynamic queue length thresholds for shared-memory packet switches. In *In Proceedings of INFOCOM*, pages 679–687, 1998.

[33] Santosh Krishnan, Abhijit K. Choudhury, and Fabio M. Chiussi. Dynamic partitioning: A mechanism for shared memory management. In *proceedings IEEE INFOCOM*, pages 144–152, 1999.

[34] Daniel Llorente, Kimon Karras, Thomas Wild, and Andreas Herkersdorf. Buffer allocation for advanced packet segmentation in network processors. *Application-Specific Systems, Architectures and Processors, IEEE International Conference on*, 0:221–226, 2008. `doi:10.1109/ASAP.2008.4580182`.

[35] Sven Wuytack, Julio L. da Silva, Jr., Francky Catthoor, Gjalt de Jong, and Chantal Ykman-Couvreur. Memory management for embedded network applications. *Readings in hardware/software co-design*, pages 465–476, 1999.

[36] Aristides Nikologiannis and Manolis Katevenis. Efficient per-flow queueing in DRAM at OC-192 line rate using out-of-order execution techniques. In *In Proceedings of ICC 2001*, pages 2048–2052, 2001.

[37] A. Nikologiannis, I. Papaefstathiou, G. Kornaros, and C. Kachris. An FPGA-based queue management system for high speed networking devices. *Microprocessors and Microsystems*, 28(5-6):223 – 236, 2004. Special Issue on FPGAs: Applications and Designs. `doi:10.1016/j.micpro.2004.03.014`.

[38] G. Kornaros, I. Papaefstathiou, A. Nikologiannis, and N. Zervos. A fully-programmable memory management system optimizing queue handling at multi gigabit rates. In *Proceedings of the 40th annual Design Automation Conference (DAC 2003)*, pages 54–59, New York, NY, USA, 2003. ACM. `doi:10.1145/775832.775849`.

[39] I. Papaefstathiou, T. Otphanoudakis, G. Kornaros, C. Kachris, I. Mavroidis, and A. Nikologiannis. Queue management in network processors. *Proceedings of the Conference Design, Automation and Test in Europe (DATE), 2005.*, pages 112–117 Vol. 3, 7-11 March 2005. `doi:10.1109/DATE.2005.251`.

[40] Ch. Ykman-Couvreur, J. Lambrecht, D. Verkest, F. Catthoor, A. Nikologiannis, and G. Konstantoulakis. System-level performance optimization of the data queueing memory management in high-speed network processors. In *Proceedings of the 39th annual Design Automation Conference (DAC 2002)*, pages 518–523, New York, NY, USA, 2002. ACM. `doi:10.1145/513918.514050`.

[41] M. Alisafaee, S.M. Fakhraie, and M. Tehranipoor. Architecture of an embedded queue management engine for high-speed network devices. In *48th Midwest Symposium on Circuits and Systems*, pages 1907 –1910 Vol. 2, August 2005. `doi:10.1109/MWSCAS.2005.1594498`.

[42] D. Llorente, K. Karras, M. Meitinger, H. Rauchfuss, T. Wild, and A. Herkersdorf. Accelerating packet buffering and administration in network processors. In *Integrated Circuits, 2007. ISIC '07. International Symposium on*, pages 373 –377, sept. 2007. `doi:10.1109/ISICIR.2007.4441876`.

[43] Intel. *Intel Internet Exchange Architecture Portability Framework Developers Manual, SDK 3.5 Release*. Intel corporation, November 2003.

[44] Sailesh Kumar, Jonathan Turner, and Patrick Crowley. Addressing queuing bottlenecks at high speeds. In *HOTI '05: Proceedings of the 13th Symposium on High Performance Interconnects*, pages 107–113, Washington, DC, USA, 2005. IEEE Computer Society. `doi:10.1109/CONECT.2005.7`.

[45] S. Iyer, R.R. Kompella, and N. McKeown. Designing packet buffers for router linecards. *IEEE/ACM Transactions on Networking*, 16(3):705–717, June 2008. `doi:10.1109/TNET.2008.923720`.

[46] Curtis Villamizar and Cheng Song. High performance TCP in ANSNET. *SIGCOMM Computer Communication Review*, 24:45–60, October 1994. doi:http://doi.acm.org/10.1145/205511.205520.

[47] Joel Sommers, Paul Barford, Albert Greenberg, and Walter Willinger. An SLA perspective on the router buffer sizing problem. *SIGMETRICS Perform. Eval. Rev.*, 35(4):40–51, 2008. doi:10.1145/1364644.1364645.

[48] Roy Rubenstein. Pushing packet performance - leading edge chipset powers 100Gbit/s IP router platform. New Electronics, January 2010. http://fplreflib.findlay.co.uk/articles/22079%5Cp27-28.pdf.

[49] Spirent Communications. http://www.spirent.com.

[50] Agilent Technologies. http://www.agilent.com.

[51] Garry Lemasa and Silvano Gai. Fibre Channel over Ethernet in the data center: An introduction (white paper), 2007. www.fibrechannel.org.

[52] Classle. http://www.classle.net/node/23951.

[53] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008.

[54] Samsung Semiconductor. http://www.samsung.com/global/business/semiconductor/.

[55] MagnaLynx. http://www.magnalynx.com.

[56] QDR Consortium. http://www.qdrconsortium.org/.

[57] GSI Technology. http://www.gsitechnology.com/.

[58] NEC Electronics Corporation. *How to Use QDR II SRAMs and DDR II SRAMs*, 1st edition, March 2008. http://www2.renesas.com/memory/en/download/M19119EJ1V0UM00.pdf.

[59] Michael Pearson. QDRIII: Next generation SRAM for networking, October 2004. http://www.qdrconsortium.org/presentation/QDR-III-SRAM.pdf.

[60] Rambus. www.rambus.com.

[61] Samsung semiconductor, DDR3-SDRAM, 2009. http://www.samsung.com/global/business/semiconductor/products/dram/Products_DDR3SDRAM.html.

[62] DDR2 SDRAM Fully Buffered DIMM (FBDIMM) design standard, JESD205, Mar 2007. http://www.jedec.org.

[63] SPMT (Serial Port Memory Technology). http://www.spmt.org/.

[64] JEDEC (joint electron devices engineering council). `http://www.jedec.org/`.

[65] Micron Technology, Inc. `http://www.micron.com/`.

[66] Ralph Schlenk and Christian Hermsmeyer. Scalable architectures for 100 GbE packet processing. In *Proceedings of the 10. ITG Symposium on Photonic Networks*, Leipzig, Germany, May 2009.

[67] GDDR5 SGRAM standard, JESD212, Sep 2009. `http://www.jedec.org/`.

[68] Micron Technology, Inc., RLDRAM. `http://www.micron.com/products/dram/rldram.html`.

[69] Rambus XDR-DRAM, 2010. `http://www.rambus.com/us/technology/solutions/xdr/index.html`.

[70] Rambus XDR2-DRAM, 2010. `http://www.rambus.com/us/technology/solutions/xdr2/index.html`.

[71] Sundar Iyer, Rui Zhang, and Nick McKeown. Routers with a single stage of buffering. *SIGCOMM Computer Communications Review*, 32(4):251–264, 2002. `doi:10.1145/964725.633050`.

[72] Cheng-Shang Chang, Duan-Shin Lee, and Ching-Ming Lien. Load balanced Birkhoff-von Neumann switches with resequencing. *SIGMETRICS Perform. Eval. Rev.*, 29(3):23–24, 2001. `doi:10.1145/507553.507563`.

[73] Hao Wang and Bill Lin. Block-based packet buffer with deterministic packet departures. In *Proceedings of the IEEE Workshop on High Performance Switching and Routing (HPSR)*, June 2010.

[74] J.M. McCollum, Xike Li, and I. Elhanany. A multistage pipelined memory management algorithm for parallel shared memory switches. In *Circuits and Systems, 2005. 48th Midwest Symposium on*, pages 1911 –1914 Vol. 2, 7-10 2005. `doi:10.1109/MWSCAS.2005.1594499`.

[75] Y.-M. Joo and N. McKeown. Doubling memory bandwidth for network buffers. In *Proceedings of the Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '98)*, volume 2, pages 808 –815 vol.2, 29 1998. `doi:10.1109/INFCOM.1998.665104`.

[76] Sundar Iyer, Ramana Rao Kompella, and Nick Mckeown. Analysis of a memory architecture for fast packet buffers. In *Proceedings of the IEEE Workshop on High Performance Switching and Routing (HPSR)*, pages 368–373, 2001.

[77] Sailesh Kumar, Patrick Crowley, and Jonathan Turner. Design of randomized multichannel packet storage for high performance routers. In *HOTI '05: Proceedings of the 13th Symposium on High Performance Interconnects*, pages 100–106, Washington, DC, USA, 2005. IEEE Computer Society. `doi:10.1109/CONECT.2005.17`.

[78] Feng Wang and M. Hamdi. Matching the speed gap between SRAM and DRAM. In *Proceedings of the IEEE Workshop on High Performance Switching and Routing (HPSR)*, pages 104 –109, May 2008. `doi:10.1109/HSPR.2008.4734429`.

[79] Mohammad Alisafaee, Shabnam Ataee, and Sied Mehdi Fakhraie. Abstract bandwidth-enhanced waste-free control technique for multi-queue network buffers. In *International Symposium on Telecommunications (IST2005)*, Shiraz, Iran, September 2005.

[80] G. Shrimali, I. Keslassy, and N. McKeown. Designing packet buffers with statistical guarantees. In *Proceedings of the 12th Annual IEEE Symposium on High Performance Interconnects, 2004.*, pages 54 – 60, 25-27 2004. `doi:10.1109/CONECT.2004.1375202`.

[81] G. Shrimali and N. McKeown. Building packet buffers using interleaved memories. In *High Performance Switching and Routing, 2005. HPSR. 2005 Workshop on*, pages 1–5, May 2005. `doi:10.1109/HPSR.2005.1503183`.

[82] Sundar Iyer, Ramana Rao Kompella, and Nick Mckeown. Designing packet buffers for router linecards. Technical report, Stanford University, Department of Computer Science, High Performance Networking Group, 2002.

[83] J. Garcia, L. Cerda, J. Corbal, and M. Valero. A conflict-free memory banking architecture for fast VOQ packet buffers. In *IEEE Global Telecommunications Conference (GLOBECOM'03)*, volume 7, pages 4158 – 4162 vol.7, Dec 2003. `doi:10.1109/GLOCOM.2003.1259010`.

[84] J. Garcia, L. Cerda, and J. Corbal. A conflict-free memory banking architecture for fast packet buffers. Technical report, Polytechnic University of Catalonia (UPC), Computer Architecture Department, Jul 2002.

[85] Jorge Garcia-Vidal, Maribel March, and Jesus Corbal. A DRAM/SRAM memory scheme for fast packet buffers. *IEEE Transactions on Computers*, 55(5):588–602, 2006. `doi:10.1109/TC.2006.63`.

[86] Feng Wang and M. Hamdi. Scalable router memory architecture based on interleaved DRAM. In *Proceedings of the IEEE Workshop on High Performance Switching and Routing (HPSR)*, pages 6 pp.–, 0-0 2006. `doi:10.1109/HPSR.2006.1709682`.

[87] Feng Wang and M. Hamdi. Scalable router memory architecture based on inter-leaved dram: Analysis and numerical studies. In *IEEE International Conference on Communications (ICC)*, pages 6380 –6385, June 2007. `doi:10.1109/ICC.2007.1056`.

[88] Feng Wang, Mounir Hamdi, and Jogesh K. Muppala. Using parallel DRAM to scale router buffers. *IEEE Transactions on Parallel and Distributed Systems*, 20:710–724, 2009.

[89] Feng Wang and Mounir Hamdi. Memory subsystems in high-end routers. *IEEE Micro*, 29:52–63, May 2009. `doi:10.1109/MM.2009.45`.

[90] Ed Clarke. FPGAs and structured ASICs: Low-risk SoC for the masses. *Design & Reuse – Industry Articles*, January 2006.

[91] Jordan Plofsky. The changing economics of FPGAs, ASICs and ASSPs. *RTC Magazine*, April 2003.

[92] Xilinx Corporation. *Virtex-6 FPGA Memory Resources*, UG363 (v1.5) edition, August 2010. `http://www.xilinx.com/support/documentation/user_guides/ug363.pdf`.

[93] Xilinx Corporation. *Virtex-6 Family Overview*, DS150 (v2.2) edition, January 2010. `http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf`.

[94] Altera Corporation. *Stratix V Device Family Overview*, SV51001-1.3 edition, July 2010. `http://www.altera.com/literature/hb/stratix-v/stx5_51001.pdf`.

[95] Institute of Communication Networks and Computer Engineering. The Universal Hardware Platform (UHP), 2005. `http://www.ikr.uni-stuttgart.de/Content/UHP/`.

[96] Altera Corporation. *Stratix II Device Handbook*, May 2007. `http://www.altera.com/literature/hb/stx2/stratix2_handbook.pdf`.

[97] Mentor Graphics Corporation – HDL Designer. Homepage, July 2010. `http://www.mentor.com/products/fpga/hdl_design/hdl_designer_series/`.

[98] Mentor Graphics Corporation – Modelsim. Homepage, July 2010. `http://www.mentor.com/products/fpga/simulation/modelsim`.

[99] Mentor Graphics Corporation – Precision Synthesis. Homepage, November 2010. `http://www.mentor.com/products/fpga/synthesis/precision_rtl/`.

[100] Altera Corporation. *Quartus II Handbook*, 10.0 edition, July 2010. `http://www.altera.com/literature/hb/qts/quartusii_handbook_10.0.pdf`.

[101] Thomas Unmuth. Entwurf und Realisierung eines Management-Systems in VHDL. Studienarbeit, Institut für Kommunikationsnetze und Rechnersysteme, Universität Stuttgart, October 2006.

[102] Oliver Refle. Entwurf und Implementierung einer Software-Architektur zur Steuerung eines Hardware-Management-Systems. Studienarbeit, Institut für Kommunikationsnetze und Rechnersysteme, Universität Stuttgart, July 2006.

[103] Alexander Schabel. Realisierung einer konfigurierbaren grafischen Bedienoberfläche in Java für ein FPGA-Management-System. Studienarbeit, Institut für Kommunikationsnetze und Rechnersysteme, Universität Stuttgart, January 2008.

[104] Arthur Mutter and Martin Köhn. Management-System for hardware designs. Presentation, November 2006.

[105] Matthew R. Pillmeier, Michael J. Schulte, E. George, and E. George Walters III. Design alternatives for barrel shifters, 2002. `http://www.princeton.edu/~rblee/ELE572Papers/Fall04Readings/Shifter_Schulte.pdf`.

[106] Mentor Graphics Corporation. *Precision Synthesis Reference Manual*, 2011.

[107] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):203 –215, February 2007. `doi:10.1109/TCAD.2006.884574`.

[802.1ah] IEEE Computer Society. 802.1ah: Draft Standard for Local and Metropolitan Area Networks–Virtual Bridged Local Area Networks, Amendment 6: Provider Backbone Bridges, November 2007.

[802.3] IEEE Computer Society. 802.3: IEEE Standard for Local and Metropolitan Area Networks–Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications, 2005.

Acknowledgments

In the years 2004 to 2010 I was a scientific stuff member at the Institute of Communication Networks and Computer Engineering (IKR) at the University of Stuttgart. In these six years I worked in the area of communication networks and digital system design. The interesting research projects I could work on and the nice colleagues and project partners I had turned this time into an exiting and great stage of my life.

At first, I'd like to thank Prof. Paul J. Kühn for the opportunity to be a research stuff member and for the supervision of this thesis. Especially in the last phase of the thesis he gave me a lot of constructive feedback and never stopped motivating me to go on. I also want to thank Prof. Andreas Kirstädter –the new institute leader– and Prof. Ernst Biersack from EURECOM in France for reviewing my thesis and joining the defense.

I want to thank all my former colleagues for the great working environment at IKR. Particular thanks go to my friends from the High Speed Networks group Sebastian Gunreben (my room mate), Simon Hauger, Martin Köhn (my mentoring colleague), Sascha Junghans, Detlef Saß, Joachim Scharf, Frank Feller, Guoqiang Hu, Elisabeth Georgieva as well as to my friends from the Hardware group Matthias Meyer and Oswin Horvath.

Many people shared the ideas of this theses with me and gave me feedback, reviewed this document, and helped me during the preparation for the oral defense. As they enabled me to write this thesis I am very grateful to these people: Simon Hauger, Oswin Horvath, Joachim Scharf, Jochen Kögel, Marc Necker, Marc Barisch, Thomas Werthmann, and Christian Blankenhorn. Finally, many thanks to Ulrich Gemkow for the professional advise and feedback on how to write a thesis.

During my time at university I worked on several bilateral projects with Altcatel Lucent Bell Labs Germany. These projects were important to me, as the brought the idea for my thesis. For the great cooperation I'd like to thank especially Gert Eilenberger, Wolfram Lautenschläger, Lars Dembeck, Jens Milbrandt, and Stefan Bunse.

Finally, I'd like to thank my wife Christine and our daughter Sarah for their patience during thesis writing and the motivation they gave me every day. I'm looking forward to spend our time together – starting now. My parents and my brother also supported me consequently. I am very thankful for this.