Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. 41

Matthias Gries

Algorithm-Architecture Trade-offs in Network Processor Design

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Diss. ETH No. 14191

Algorithm-Architecture Trade-offs in Network Processor Design

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
Doctor of Technical Sciences

presented by
MATTHIAS GRIES

Dipl.-Ing., TU Hamburg-Harburg, Germany
born May 15, 1972
citizen of Germany

accepted on the recommendation of
Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Wolfgang Fichtner, co-examiner

2001

Examination date: May 21, 2001

Abstract

The increasing use of computer networks for all kinds of information exchange between autonomous computing resources is associated with a number of side-effects. In the Internet, where computers all over the globe are interconnected, the traffic volume grows faster than the infrastructure improves, leading to congestion of networking routes. In the application domain of embedded systems, networks can be used to couple complex sensor systems with a computing core. The provision of raw bandwidth may not be sufficient in such systems to allow control with real-time constraints. The underlying requirement in both cases is a network service with a defined quality, for instance, in terms of traffic loss ratio and worst-case communication delay. The provision of suitable communication services however requires a noticeable overhead in terms of computing load. Therefore, application-specific hardware accelerators – so-called network processors – have been introduced to speed up or even enable the maintenance of certain network services. The following issues have not yet been dealt with:

- Although there are network processors for high-speed networks, no processor is available that considers the requirements of the interface between networks of a service provider and a customer.

- While each individual task of a network processor is well understood, it is unclear how different tasks, that potentially show interfering properties, should cooperate to preserve the service quality.

The above issues are addressed in this thesis and the major contributions in the research area of algorithms and architectures for network processors are:

- A service scheme is defined which takes care of the requirements at the interface between networks of a service provider and a customer.

- Various combinations of network processing tasks are explored for this service scheme by exhaustive simulation. The exploration focuses on the preservation of service quality parameters.

- The exploration of processing tasks is combined with an evaluation of suitable building blocks for architectures of network processors so that the interaction of algorithm behavior and timing of hardware resources can be examined by co-simulation of both aspects.

- Due to its impact on the overall performance, refinements of a particular building block – the memory controller – are evaluated. The influence of an application-specific memory controller is explored by extensive simulation of various benchmarks, dynamic RAMs, and memory access schemes incorporated into a mature CPU simulator.

Kurzfassung

Aufgrund der zunehmenden Verbreitung von Computernetzwerken für den Informationsaustausch aller Art zwischen autonomen Rechnern kann man gewisse Seiteneffekte beobachten. Im Internet, das Rechner verteilt auf der ganzen Welt miteinander verbindet, steigt die Menge des Verkehrs stärker an, als Netzwerkinfrastruktur nachgerüstet werden kann. Dies führt zu einer Überlastung von Netzwerkverbindungen. Ein weiteres Beispiel sind eingebettete Systeme, in denen Netzwerke eingesetzt werden können, um Sensorsysteme an einen Rechnerkern zu koppeln. Es ist in solchen Systemen meistens nicht ausreichend, lediglich Durchsatz durch ein Netzwerk zur Verfügung zu stellen, da ggf. Echtzeitbeschränkungen nicht eingehalten werden können. Beide Anwendungsfälle haben gemeinsam, dass Dienste definierter Qualität benötigt werden, z.B. spezifiziert gemäss einer Verlustrate oder einer maximalen Verzögerung. Die Unterstützung von geeigneten Kommunikationsdiensten bringt jedoch einen merklichen Anstieg an Rechenzeitbedarf mit sich. Deshalb sind anwendungsspezifische Hardwarebeschleuniger – sog. Netzwerkprozessoren – vorgeschlagen worden, um Netzwerkdienste zu etablieren und zu ermöglichen. Die folgenden Fragestellungen sind bisher noch nicht behandelt worden:

- Obwohl viele Netzwerkprozessoren für hohen Durchsatz erhältlich sind, fehlen Prozessoren, welche die Bedürfnisse an der Schnittstelle zwischen Netzwerken eines Dienstanbieters und eines Kunden berücksichtigen.

- Während die einzelnen Aufgaben eines Netzwerkprozessors umfassend verstanden sind, ist nach wie vor unklar, in welcher Form mehrere Funktionen, deren Eigenschaften sich möglicherweise gegenseitig beeinflussen, zusammenarbeiten müssen, um eine Dienstqualität beizubehalten.

Diese Punkte werden in der vorliegenden Arbeit behandelt, und die Hauptbeiträge zum Forschungsgebiet der Algorithmen und Architekturen von Netzwerkprozessoren sind:

- Eine Abstufung von Netzwerkdiensten wird vorgestellt, welche die Anforderungen an der Schnittstelle zwischen Kunden- und Dienstanbieternetzwerken berücksichtigt.

- Verschiedene Kombinationen von Funktionen, die ein Netzwerkprozessor ausführen muss, werden für die vorgeschlagene Abstufung von Diensten mit

Hilfe von umfangreichen Simulationen untersucht. Dabei wird sich auf die Beibehaltung der Dienstqualität konzentriert.

- Die Untersuchung der Funktionen wird mit einer Bewertung geeigneter Architekturblöcke für Netzwerkprozessoren kombiniert, so dass der Einfluss von Algorithmen auf die Auslastung von Hardwareressourcen durch gleichzeitige Simulation beider Aspekte ergründet werden kann.

- Die Diskussion eines bestimmten Hardwareblocks, des sog. Speichercontrollers, wird weiter verfeinert, da dieser Block grosse Auswirkungen auf die Gesamtperformanz eines Systems zeigt. Der Einfluss eines anwendungsspezifischen Speichercontrollers wird durch umfangreiche Simulationen verschiedener Testanwendungen, dynamischer RAMs und unterschiedlicher Speicherzugriffsverfahren aufgezeigt, die in einen ausgereiften CPU-Simulator integriert worden sind.

I would like to thank

- Prof. Lothar Thiele for providing a pleasant research environment, for advising my research work, and for the assistance during my thesis as well as Prof. Wolfgang Fichtner for his support as a co-examiner of my thesis,
- my colleagues Marco Platzner, Jan Beutel, Jonas Greutert, and Samarjit Chakraborty for valuable discussions and suggestions to improve my work as well as for carefully proof-reading my thesis,
- my colleagues Rob Esser, Jörn Janneck, Kim Mason, and Martin Naedele, who are the authors of the modeling and simulation tool-set MOSES/CodeSign [46, 90, 4], for their personal assistance in using their excellent software.

Contents

1

Introduction

The access and exchange of information is more and more being driven by the globally available Internet that interconnects various kinds of autonomous computing resources. Moreover, computer networks are increasingly being used to link several distributed computers for scientific calculations and other computations. Due to the tremendous growth rate of the Internet and an increasing spread of networking technology in various application domains, e.g. in embedded systems, the objectives of network design are more and more shifting from providing raw bandwidth to providing services with a defined quality. The enhanced functionality needed by nodes of the network to adequately support services requires application-specific hardware accelerators – so-called network processors. These processors are designed to relieve the main computing resources from service tasks and to enable additional services.

An outline of the functionality of a computer network is given in Section 1.1 and some specific terms are introduced which will be used throughout the thesis to discuss issues related to networks. Section 1.2 motivates the research topics that are addressed by this work. It is shown that there is a lack of work related to suitable algorithms and architectures for network processors that are targeted on the interface between networks of a customer and a service provider. In addition, the contributions of this thesis are discussed. Finally in Section 1.3 an overview of the remaining chapters is given.

1.1 Computer networks

During the early stages of building the Internet the main goal was to set up a decentrally organized interconnection of computers with redundancy so that a breakdown of a part of the network would not affect the connectivity and efficiency of the overall Internet. Consequently, a node of the network only knows its neighbors which are identified by unique addresses. Since routes through the network are not determined statically but dynamically depending on the current state of the network, every single data entity must be processed by every intermediate network node from the source to the destination of a transmission. Packets from different transmissions should be treated equally by the network. Nodes make a "best effort" to handle all packets in the order of their arrival.

In the following, basic background information is provided that shows how the communication between two computers over a network is organized so that the information exchange is transparent for applications. Moreover, some ambiguous network terms used in this thesis are clarified. The next section will then motivate the focus of this thesis.

1.1.1 Communication layers

The tasks involved with the end-to-end communication of two network nodes can be divided into a set of abstract functionalities, called *layers*, that form a hierarchy. Each layer can only pass information to the next higher or lower layer through defined interfaces. At each layer, protocols define the operations and responses necessary to exchange information between peer layers at different network nodes. This information is held by layer-specific header fields that are added to traffic entities. Lower layers only consider the transmission of traffic between neighboring network nodes whereas the higher layers affect the end-to-end transmission through several intermediate nodes. The Open Systems Interconnection (OSI) reference model by ISO is composed of seven abstract layers. The Transmission Control/Internet Protocol (TCP/IP) stack used by the Internet only considers five of these layers. The reader is referred to further introductory literature – e.g. [157] – for a more detailed discussion of network layers. The TCP/IP protocol stack distinguishes the following layers[1], see Fig. 1:

- *Physical layer:* This is the lowest layer. It considers the plain transmission of data streams through a physical medium, e.g. a copper wire, between neighboring nodes.

- *Link layer:* This layer is responsible for reliable transmission of traffic entities (frames) between neighboring nodes.

- *Network layer:* This is the lowest layer that affects the plain end-to-end transmission of data packets. At this level, the actual navigation (routing) through

[1] The original TCP/IP reference model in [29] does not include any description of a physical or a link layer. These two layers have been added from the OSI reference model to show a complete hierarchy of layers as it is used in the Internet.

the network is enabled.

- *Transport layer:* The transport layer is responsible for end-to-end transmission of aggregated packets, so-called segments or messages. A reliable transmission may be enabled by packet sequencing and flow control.

- *Application layer:* This layer deals with the exchange of data between applications running at different network nodes. For instance, the FTP protocol handles whole file transfers and the HTTP protocol is responsible for web page downloads.

The different layers are often numbered, beginning with the lowest layer.

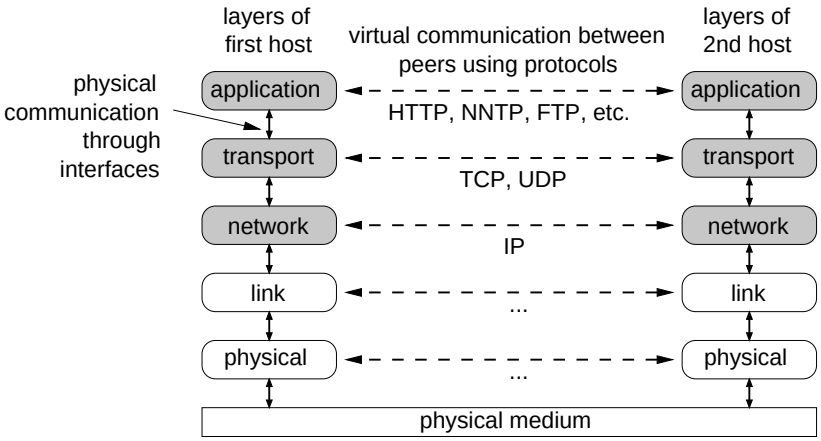

Fig. 1: Communication layers used by the TCP/IP protocol stack with interfaces and protocols. The grey-marked layers affect the end-to-end transmission of information between two nodes along a route through the network whereas the white layers only influence the transfer to the next node.

1.1.2 Terms and definitions

Communication networks can be categorized according to different criteria.

- *Classification according to geographic coverage:* A *Local Area Network (LAN)* interconnects end devices (workstations, printers, etc.) within a relatively small area, e.g. bounded by a room, a building, or some buildings belonging to the same institution. In the latter case, LANs are also called campus area or enterprise networks. Almost always some shared medium is used for communication and no explicit routing is required between nodes of the same LAN. A *Wide Area Network (WAN)* interconnects LANs that are possibly based on different technologies and may belong to different organizational units spread over a large geographic area. Explicit routing is needed to find a path between LANs. A WAN network with intermediate extension limited to a town or a city is also called a *Metropolitan Area Network (MAN)*.

- *Classification according to connectivity:* Especially when talking about the Internet, a hierarchy of networks is introduced. Routers under a single technical administration and a single routing policy are interconnected to an Autonomous System (AS) which aggregates a group of address prefixes. The interconnection of autonomous systems forms the Internet. An autonomous system will belong to the highest (outermost) level of a network, the so-called *access network*, if customer links enter the Internet via this autonomous system. The first router seen by the customer's traffic is the edge router of an Internet Service Provider (ISP) running the autonomous system. The remaining interconnection of autonomous systems without the network edge constitutes the lowest network level, the *core* or *backbone network*. One can define an additional level between the access and the core network, called *distribution network*. Whether an autonomous system belongs to the backbone or to the distribution network is determined by the number of connections to other autonomous systems [47]. Autonomous systems in the distribution network use less connections to other systems than autonomous systems in the core. The distribution network provides the transit between the access and core parts of the network.

The following clarification of network terms is included to unambiguously determine their meaning in the context of this thesis.

Def. 1: **(Router)** *A router is a processing stage of a network node that determines the next node to which a packet should be forwarded in order to reach the destination. The router is therefore connected to several nodes. It individually decides for each incoming packet based on the current state of the network which way the packet should be sent. Routers operate at the network layer.*

Def. 2: **(Hop)** *A* hop *is an intermediate network connection between two routers over which a packet is transmitted to reach its destination.*

Def. 3: **(Flow)** *A* flow *is a sequence of packets passing a network node. The packets of a flow are similarly treated by the node with regard to routing and other policies. That means, each processing stage within a node only uses a single setting to handle the packets of a flow. A flow may be an aggregation of packets from different applications or transport layer sessions which are the subject of the same service requirement.*

Def. 4: **(Service)** *A* service *may range from the provision of plain network access to the support of certain protocols and applications such as e-mail, video streaming, or voice telephony.*

Def. 5: **(Quality of Service (QoS))** QoS *is a performance specification that covers the properties of a service for a single flow or a whole class of flows. QoS may be specified by parameters such as data loss ratios, delay and throughput guarantees, delay characteristics (jitter), etc.*

Def. 6: **(Service Level Agreement (SLA))** *An* SLA *is a contract between a network service provider and a customer that specifies, usually in measurable terms, what services with which QoS the provider will offer for the customer. Besides the description of QoS parameters and assigned flows an SLA may also include specifications of the network availability, the number of concurrent users, etc.*

1.2 Design challenges

Looking at the current growth rate of the performance of networking, computing, and volatile storage technology, a diverging development can be recognized. The random access time of dynamic RAMs only halves approx. every ten years [126]. Contrary to that, the computing performance of CPUs doubles every 18 to 24 months [131, 146, 138]. This is why RAM resources have become the major performance bottle-neck of computing systems [24, 18, 167]. Moreover, although the maximum link bandwidth used in the Internet increases at almost the same speed as computing performance, the volume of Internet traffic currently doubles every six months [131]. Therefore, packet processing tasks will no longer be performed by general computing resources but must be accelerated by application-specific network processors. The lack of computing and RAM resources for networking motivates the discussion of the following two research areas.

1.2.1 Efficient network processing

The current growth rate of the Internet leads to congestion of major parts of the network since the infrastructure cannot be updated at the same speed. As a result, degraded connectivity and even starvation of transmissions are appearing. Moreover, certain flows may occupy more networking resources than others because nodes usually handle packets without considering any flow-specific information. Consequently, a flow may greedily use bandwidth by, for instance, transferring only relatively large packets. Therefore, the access to networks must be regulated according to reservations and the network must be protected against greedy flows. A network node has to apply more sophisticated methods than best effort to maintain the QoS for customers. Algorithms are required to affect the packet processing starting at the network layer from which end-to-end transmissions are distinguishable. This thesis will hence elaborate on packet processing tasks at the network layer. In particular, the following points are addressed:

- The end-to-end QoS preservation through a core network that only handles aggregates of flows depends on flow classification and policing performed at the edge of the network. Hence, this thesis will focus on proficient packet processing at the access network.

- Related works only investigate single packet processing stages. It is therefore shown how the cooperation of policing, specific queuing, and packet scheduling can actually be used to preserve and guarantee QoS requirements of a service level agreement.

- Moreover, a new service scheme is introduced that considers the requirements of multi-service access networks.

- Although processors for distribution and core networks are available that support QoS distinction, no processors with such facilities can be found for the requirements of the access network edge. An exploration of suitable architectures together with varying combinations of algorithms is thus performed by co-simulation of algorithm behavior and hardware timing of selected building blocks. In this way, algorithm behavior, hardware resource load, and QoS properties are evaluated together for a network processor application.

1.2.2 Data handling in network nodes

A network processor has to manage a variety of data objects with different characteristics. There are traffic streams that must be buffered. As more flows are distinguished, the access patterns of the buffer memory become more uncorrelated [98] so that caches cannot effectively be employed. Moreover, each packet processing stage uses some local variables and parameters. We will see that the use of caching is prohibitive for processing stages until the QoS context information can be deduced for a packet. Processing stages that could potentially employ additional caches for parameters and values however suffer from a lack of access locality due to possibly random flow variations from packet to packet. This is why all currently available network processors with QoS distinction rely on several separate memory areas of different technology and do not use caches. We will therefore have a closer look at the exploitation of different RAM resources in this thesis. The design space exploration for network processors is refined by an exploration of memory access schemes applying different benchmarks and RAM types. The exploration underpins the impact of an adequate memory controller that should be integrated into a network processor as an application-specific circuit. The following contributions of this thesis towards the efficient utilization of RAM resources can be stated:

- Based on an analysis of RAM architectures a memory controller model is derived in a visual formalism which takes advantage of internal parallel hardware blocks of dynamic RAMs. In this manner we graphically document the properties of the inner architecture of a memory controller more precisely than current data sheets do.

- Based on the insight gained by the preceding analysis, performance models of different memory controllers and DRAMs are added to a mature CPU simulator. An exploration of computing performance is performed by simulation of various applications, memory controller access schemes, and DRAM types. In this

way we are able to show how heavily the performance of an embedded system such as a network processor depends on the chosen memory controller access scheme.

1.3 Overview

This thesis contributes to the design of application-specific network processors that relieve a main computing system from packet processing tasks such as preserving Quality of Service (QoS) for traffic flows. It is focused on access networks where a customer's traffic enters the network of a service provider. Algorithms and architectures for suitable network processors supporting a new service scheme are explored and evaluated. This work is structured as follows:

- ***Chapter 2*** introduces packet processing tasks at the network layer that are candidates for acceleration by network processors in the common Internet. Related work for each task is discussed and available service schemes are presented. Moreover, a method based on service curves is shown which simplifies the determination of Quality of Service (QoS) requirements.

- ***Chapter 3*** continues with an introduction of a new service scheme. Packet processing tasks that are responsible for preserving the QoS are adapted according to the needs of the new service scheme. A design space exploration of network processors aimed at access networks is performed by co-simulating different combinations of packet processing tasks and hardware resources. Solutions described in the preceding and the following chapter are incorporated into the simulation models for algorithms and hardware blocks. In this manner, algorithm behavior and hardware load are evaluated together.

- ***Chapter 4*** describes the properties of current RAM architectures and motivates the RAM timing model used for the evaluation in the preceding chapter. A memory subsystem containing a decent memory controller is modeled by a visual formalism to graphically analyze advantageous memory access patterns that are supported by the controller. Access schemes derived from the analysis are integrated into a mature CPU simulator to perform an exploration of DRAM types and memory controller features by simulating the execution of various applications. The exploration underpins the large impact of a memory controller on the computing performance and motivates the integration of an application-specific controller into a network processor.

- ***Chapter 5*** concludes with a summary of the main results of the thesis and provides some starting points for further research.

2

IP packet processing: Requirements and existing solutions

In this chapter, packet processing tasks are described which enable and influence the Quality of Service (QoS) experienced by a flow. Related work and existing service schemes are discussed which are used in the common Internet. A methodic framework based on service curves is presented that eases the understanding of QoS requirements. The given insight into existing solutions will be used as a basis for the motivation and evaluation of our own service scheme in Chapter 3.

The network layer is the lowest layer in the OSI reference model that concerns end-to-end transmission of data. Its job is therefore to deliver packets where they are supposed to go. Reaching the destination may require to hop from network node to network node and to find a route to the destination. In the TCP/IP reference model, the Internet Layer with its Internet Protocol (IP) plays the role of the network layer. For each incoming IP packet a network node must decide to which node the packet will be forwarded next. The decision is based on the information stored in the IP packet header and additional state information in the node itself. In order to make a routing decision several tasks are involved that we call *packet processing* all together. Packet processing includes parsing the packet header, classification of the packet so as to assign a packet to a Quality of Service class, determination of the next hop (forwarding), check of Service Level Agreements (policing), queuing, and finally link scheduling, see Fig. 2. Whether all tasks are required and how complex they may become depends on the services that the network node wants to provide.

By parsing the header of an incoming packet, information about the packet becomes available for later processing such as the length of the packet, the destination address, and the protocol type. A filter stage with a small set of rules

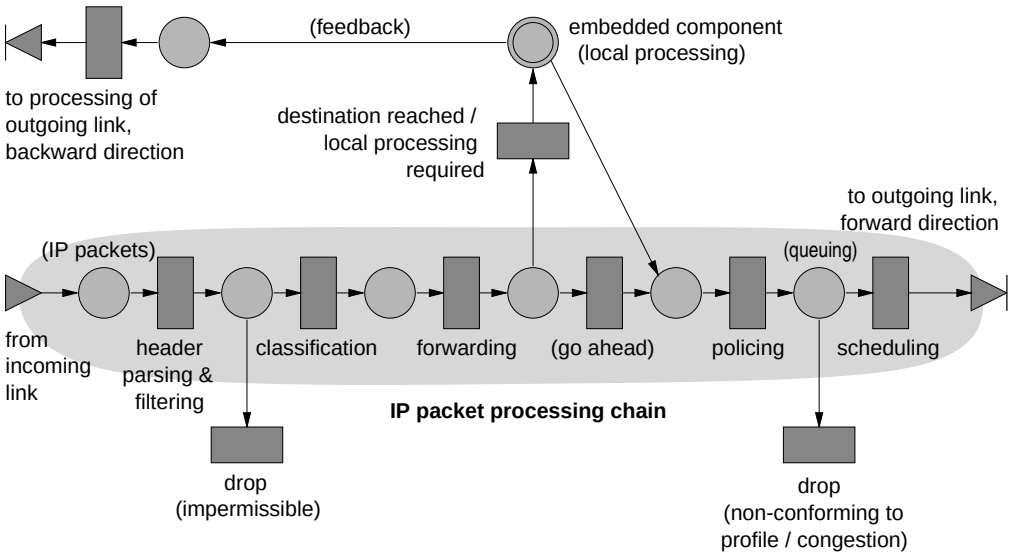

Fig. 2: The packet processing chain for the forwarding path in the Internet layer, modeled by a Petri net.

then decides based on the extracted header information whether the packet is allowed to pass further processing stages or whether it should be dropped immediately. In case of admission, a classification stage uses the extracted header fields to associate the packet with its context information such as the corresponding Quality of Service (QoS) class – a flow identifier – and the reserved rate. A forwarding stage which may be combined with the classifier uses the destination address of the packet to determine whether the packet should be passed on to a particular outgoing link or to further internal processing tasks. Further internal processing may be required when the destination is reached or some higher protocol layers must be processed. If it is decided to forward the packet to an outgoing link, the packet will be handed to the policer. The policer uses the context information assigned by the classifier to check whether a packet complies to a defined traffic profile of the corresponding flow. A traffic profile may specify properties like the maximum burstiness and the rate of incoming traffic. A profile is subject of a Service Level Agreement (SLA) between a customer and a service provider. An SLA states that as long as traffic complies to a profile, the provider will ensure a certain level of service, for instance, in terms of delay and loss. Thus, the profiler marks a packet as conforming or as non-conforming to a flow's profile. Non-conforming packets may be immediately dropped. Before the packet can finally be transmitted through the outgoing link, it must be queued until the link scheduler chooses the packet for transmission. The policy by which the scheduler chooses packets for transmission may depend on the header and context information assigned to the packet such as the packet length, the reserved rate, and the assigned QoS class.

The different packet processing tasks are introduced in the next section. In Section 2.2 the tasks which are responsible for preserving the quality of the ser-

vices during congestion and among greedy-behaving traffic are discussed. It is pointed out how the tasks cooperate so as to be able to guarantee delay bounds for packet delivery. Section 2.3 continues with a description of forwarding services for today's networks.

As introductory reading about networks in general Tanenbaum's book [157] is the first choice. The background, the enabling technologies, and the motivation for the distinction of Quality of Service (QoS) are described in Ferguson and Huston's book [52] and some further articles [73, 168, 16, 169]. Finally, discussions of forthcoming standards concerning the Internet and documentations of current standards can be found at the Internet Engineering Task Force (IETF [2]).

2.1 Packet processing tasks

2.1.1 Header parsing

Problem statement: Fields from the packet header must be extracted because their contents decide how the packet will be processed. Fields may include checksums, source and destination addresses, protocol specifiers, type of service fields as well as the length of the packet. The parsing need not be limited to the network layer header, but may also comprise headers of other OSI layers. For instance, IP-based routers often look at the source and destination port numbers of the transport layer in order to refine the classification of packets.

Although most of the required fields can be found at fixed offsets from the header start address, the parsing process becomes difficult by variable length headers which additionally use some optional fields, e.g. fields that determine secure handling or strict routes through the network. Moreover, since header parsing must be executed for all incoming packets, performance constraints may necessitate the concurrent treatment of different parts of the header although it is not clear at the beginning whether all the gathered information will be used at all. The level of parallelism is however bounded by decisions that can only be made after having read particular fields and which then require further fields from the same or another layer header.

2.1.2 Classification and routing

After having parsed the packet header, field information is available by which the incoming packet can be characterized, such as the source and destination addresses. The header fields are used to assign context information to a packet, for instance, the corresponding QoS class and the outgoing link to the next hop.

In detail, a packet may pass the following processing stages to be classified and routed:

- *Filtering:* Since every incoming packet is examined in this stage, only a small number of rules are applied. The rules assure that only authorized packets pass

through the following processing elements and that packets, which are directed to the current node, are taken out of the traffic stream. The latter task can also be performed by the forwarding stage.

- *Classification:* This stage resembles the filtering stage though a higher number of rules are usually applied. Context information is assigned to a packet depending on the header fields and according to a set of rules. Accounting and billing facilities as well as QoS-aware packet handling are thus enabled.

- *Forwarding:* In this stage, the actual routing decision takes place. Using the destination address the outgoing link to the next hop is determined.

Although a complete classification of a packet including filtering and routing could be executed by a single hardware stage or a single software process, the functionality is usually shared out among different sequential tasks since not all classification results are of interest for every incoming packet.

Diverse services may directly be provided by the classification stages or enabled for further processing by tasks inside or outside the packet processing chain, including:

- *Access control:* The network node may act as a firewall by blocking certain flows or traffic classes in order to prevent unauthorized use of network resources. This task is performed by the filtering stage.

- *Load balancing:* Traffic may be distributed among different routes and/or web servers. An appropriate routing protocol is responsible to adapt the routing tables of the forwarding stage accordingly.

- *Network address translation:* For instance, addresses of a virtual private network (VPN) must be converted into addresses of the public Internet. The corresponding addresses may be assigned to routing entries or classification rules.

- *Quality of service (QoS) differentiation:* Real-time traffic may be isolated from elastic and best-effort traffic. Traffic classes are processed with different priorities. Further traffic class refinements may concern user specific information. This task is performed by further stages of the packet processing chain that rely on the context information determined by the classifier.

- *Accounting and billing:* Traffic statistics are gathered for network engineering, for checking service level agreements and reservations as well as for billing customers according to the current network load and their actual traffic profile. This functionality could be integrated with the policer stage of the packet processing chain. The policer uses the context information derived by the classifier.

- *Policy-based routing:* For instance, secure communications should only pass network nodes of trusted and reliable service providers. An appropriate routing protocol is responsible to adapt the filter, classification, and routing rules of a node accordingly.

Packet classification and routing algorithms are usually evaluated by the following parameters:

- *Search time:* Time needed in order to look up the associated context information of a packet. Often, the search time is bounded by the number of required memory accesses since computation delays can frequently be neglected compared with memory access delays.
- *Storage space requirements:* The amount of memory needed for saving the lookup data structure.
- *Update time:* Time required to incrementally update the lookup data structure when a classification or routing rule must be inserted or deleted.

Note that algorithms which rely on caching and/or queuing are prohibitive for the classification and routing tasks because caching and queuing are methods to optimize the average case. The behavior of cache-based designs would heavily depend on traffic characteristics. Thus, the slowest path of the system architecture could be responsible for the occurrence of congestion and packet dropping. In this situation however, the router is forced to apply some congestion control and queuing without any context knowledge such as quality of service information. Therefore, any queuing delays are only acceptable after the classification. Consequently, classification and routing algorithms should be optimized for the worst-case and work at wire speed.

2.1.2.1 Forwarding

Problem statement: Since the Classless Inter Domain Routing (CIDR [129]) scheme has been introduced in 1993, routes are defined by the address of the destination network which is specified by an address prefix plus a prefix length. In this way, destination addresses which share the same prefix can be aggregated into a single routing table entry. The search for the next hop can then be performed by finding the longest matching prefix among N routes in the routing table in two steps. First, the set of prefixes is determined that match the given destination address of the IP packet. Then, among these prefixes, the longest prefix is selected. The packet is forwarded to the next hop that is assigned to the longest prefix.

Ex. 1: **(Forwarding by longest matching prefix search)** *Without restriction of the general applicability of the CIDR scheme 8 Bit addresses are used in the example whereas IPv4 employs 32 Bit addresses. Suppose a router uses the following routing table:*

network address (hexadecimal)	prefix length	informal (binary mask)	next hop
0	*0*	$*$	*(default route)*
20	*3*	$001*$	*link 1*
20	*4*	$0010*$	*link 2*
24	*5*	$0010\ 1*$	*link 3*
52	*7*	$0101\ 001*$	*link 2*

An incoming packet with the destination address 23_h *(*$0010\ 0011_b$*) matches the prefixes* $20_h/3$ *and* $20_h/4$ *as well as the prefix entry for the default route. Among these matches,* $20_h/4$ *is the longest prefix. Therefore, the packet will be forwarded to the next hop connected to link 2.*

Forwarding algorithms are usually discussed by assuming a backbone-like router environment, i.e., a routing table with several 10000 entries is used. Unless otherwise stated, the following comparison of algorithms only considers IPv4 destination address lookups (32 Bit addresses). The complexity of updates of the routing data structure is not attached much significance in most of the papers since it is generally agreed that forwarding tables do not have to be updated for every route change but a decent number of route changes can be bundled into a single table update. That means, table updates may occur in intervals of minutes.

Overview: A Patricia trie [114] is a general data structure that has been used for forwarding tasks in a number of software implementations. It is also often used as a basis of further specializations. Forwarding algorithms must trade off fast search times for small memory footprints. An extreme is to precompute the largest reasonable lookup table in order to find a match with a small number of memory accesses. This approach is presented by Gupta et al. [74]. The other extreme is to compress the data structure as much as possible at the expense of a higher number of memory accesses as it is done by Degermark et al. [43]. More balanced solutions are level compressed tries [119], multi-way search [102], and subtrie compression [159]. Since most of the approaches are based on observations of IPv4-based routing tables and hence depend on the characteristics of the distribution of network addresses, these data structures cannot adequately be used with IPv6. The binary search of prefix lengths illustrated by Waldvogel et al. [161] is a considerable exception since it allows to implement IPv6 lookups at a feasible complexity without depending on assumptions of the address distribution. Finally, content addressable memories provide a general hardware platform to implement search and match tasks and can consequently be used to execute forwarding lookups [108].

- ***Patricia tries:*** Many variations of this basic data structure described in [114] are used in software implementations of routing stages by Unix kernels [143]. The name trie is derived from the word retrieval. Tries are basically tree-like data structures. However, the sequence of bits or characters of the value or name to be searched is directly used to navigate through the tree by selecting a branch in each level of the tree. In the worst-case, Patricia tries are thus as bad as general binary trees. However, a Patricia trie allows to skip unpopulated levels of the tree by specifying the index of the bit or character of the argument to use for addressing the next branch. In every step through the levels of the tree, a bit of the IP destination address decides whether the left or right child node must be chosen. When a leaf node of the tree is reached, a comparison with the prefix assigned to that leaf must be performed. If the destination address of the current packet does not match the prefix, the packet can either be forwarded to a default

router or one backtracks up the trie to find a more general prefix[1]. The search and the update time can be bounded by $O(ipaddr)$ where $ipaddr$ is the length of an IP destination address (32 Bit for IPv4). The storage space requirements are determined by $O(N)$ where N is the number of routes in the data structure.

- ***Large precomputed tables:*** The work presented by Gupta et al. in [74] minimizes the number of memory accesses. In the worst-case, only two accesses are needed. This result is however obtained at the expense of memory space by precomputing an up to 24 Bit-wide prefix table. More than 30 MBytes of RAM are then needed. This approach is motivated by the observation that prefixes of current backbone routing tables such as the often used MAE-EAST data sets [111] are hardly ever longer than 24 Bits. Moreover, by using DRAM memory, the storage of precomputed tables can be kept cheap. Unfortunately, moderately efficient updates of the routing table require further extensions to the data structure and may still demand several hundred memory accesses per update.

- ***Multi-level dense data structure:*** Contrary to the preceding approach, Degermark et al. [43] use a dense data structure for storing the routing table completely in on-chip RAM by varying the length and the nesting level of pointer fields in dependence on the distribution of prefixes. The storage requirements have $O(N)$ complexity. Measurements with up to 40000 routing entries show that the data structure only requires about twice as much memory as it would be needed to just store all the prefixes. However, more than ten memory accesses may be needed to find the matching routing entry. Updates are performed by rebuilding the data structure and thus require $O(N)$ operations.

- ***Level Compressed (LC) tries:*** LC tries applied to routing tables [119] show two advantageous properties. On the one hand, weakly covered regions of a binary prefix trie can be compressed using skip values along the branches, similarly to Patricia tries. On the other hand, completely occupied subtries can be converted into efficient array substructures. However, in order to estimate the actually required number of memory accesses and the memory space, further knowledge about the address prefix distribution is required. Experimental results with up to 40000 routing entries show that LC tries roughly require twice as much memory space as Degermark's solution in favor of halving the number of memory accesses. Again, updates are implemented by rebuilding the data structure requiring $O(N)$ operations. Another very similar approach using variable length subtables is presented in [85].

- ***Multiway and multicolumn search:*** Lampson et al. [102] make use of cache lines in order to store efficient representations of subtries. Again, further information about the address prefix distribution is required to derive worst-case bounds for the memory usage. Using 32 Byte cache lines, measurements with up to 40000 routing entries show results for the search time and the storage

[1] Backtracking may also lead to the default router which is usually assigned to the root node of the trie.

requirements which are comparable with LC tries. Updates are performed by rebuilding the data structure with complexity $O(N)$.

- ***Representing compressed subtries by fixed-sized pages:*** The data structure described by Tzeng et al. in [159] is well suited for the estimation of worst-case bounds. The bounds only depend on the number of routing prefixes typically stored in today's biggest routing tables and not on the distribution of the prefixes. Nevertheless, the obtained bounds are competitive compared with the bounds of the other cited papers. Moreover, a prefix trie compression scheme is introduced which allows the efficient storage of a subtrie in fixed-sized memory segments. Tzeng et al. use a 17 Bit precomputed prefix table. A table entry points to a binary prefix trie which is partitioned into subtries of a minimal depth five and a maximal number of 31 nodes (the depth corresponds to the number of levels). Each node of such a subtrie has been encoded by three Bits. A subtrie is fetched by a single memory access. Therefore, at most five memory accesses are necessary to access a routing table entry: one access for the precomputed table, at most three accesses for the subtries, and one final access for the routing information. The storage requirements are $O(N)$ and an update of the data structure requires rebuilding it.

- ***Binary search on prefix lengths:*** Waldvogel et al. [161] also determine worst-case bounds independent of the distribution of address prefixes by performing a binary search on prefix lengths with complexity $O(\log ipaddr)$ where $ipaddr$ is the length of an IP destination address. However, the underlying data structure is based on perfect hashes. Storage as well as update time requirements thus heavily depend on the chosen hash architecture. The resulting bounds for IPv4 lookups are usually worse than the bounds presented in the other cited papers. The strength of Waldvogel's scheme however lies in its scalability with the address length. In this way, it may be well suited for IPv6 address lookups.

- ***Using Content Addressable Memories (CAMs):*** CAM memories implement a map data structure. They are able to compare a given value with all keys in a stored set concurrently. Ternary CAMs allow "don't care" bits enabling prefix matches by masking bits [108]. Thus, matches can be found in $O(1)$ time. However, updates require $O(N)$ operations. Current CAMs only permit small routing tables with up to some thousand entries. Moreover, access times are rather slow compared with current RAM technology.

2.1.2.2 Filtering and classification

Problem statement: The classifier determines the flow an incoming packet belongs to looking at one or more fields of the packet header. The classifier employs a set of N rules, each rule consisting of d ranges corresponding to d header fields (so-called dimensions). A range specifies an interval of valid values of the corresponding header field. A cost or priority value is assigned to each rule. A packet will match a rule if, for all dimensions, the field value lies in the corresponding range of the rule. Ranges in different rules are allowed to overlap,

i.e., a packet may match several rules. Thus, the classification problem is to determine the least cost / highest priority rule which applies to the packet. In the TCP/IP case, the most common fields are IP source and destination addresses, port numbers of the source and destination applications as well as the protocol type and its associated flags. One may also think of using additional information for the classification task that does not belong to the packet header such as the incoming link of the packet or the current system state, e.g., the time-of-day.

Ex. 2: **(Two-dimensional classification)** *The source and destination addresses of a packet are used for the classification in this example. The addresses are specified by 8 Bit values. The router uses the following set of rules which is also displayed in Fig. 3:*

Rule	X range (source addr.)	Y range (destination addr.)	priority	flow and context info.
R1	*20 - 70*	*20 - 200*	*20*	*multi-media*
R2	*150 - 180*	*70 - 110*	*10*	*CBR video*
R3	*40 - 220*	*50 - 120*	*30*	*video*
R4	*110 - 190*	*140 - 210*	*5*	*voice*
R5	*0 - 255*	*0 - 255*	*100*	*best-effort*

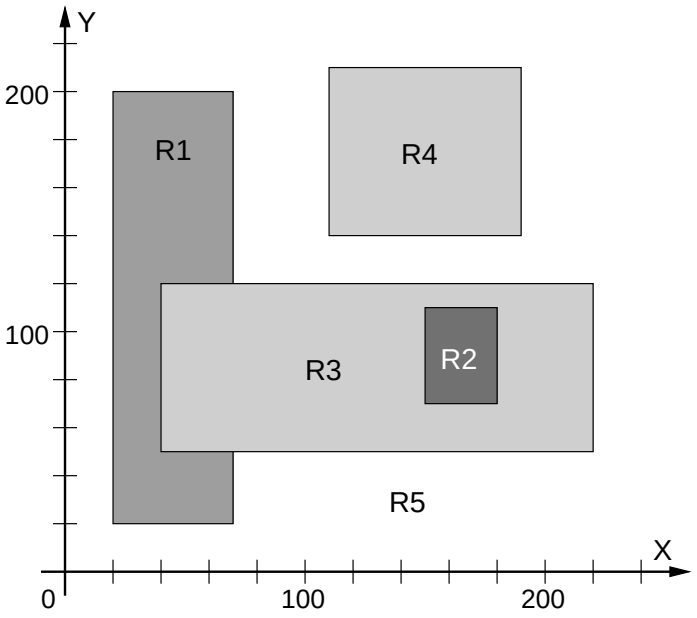

Fig. 3: Exemplary two-dimensional rule set of a packet classifier.

Ranges may overlap and a rule may even be completely covered by another rule. An incoming packet with source address 60 and destination address 80 matches the rules R1, R3, and R5. Rule R1 has the highest associated priority (lowest priority value) and the packet is thus classified to belong to the class of multi-media traffic.

Although it is agreed in stating that updates of the rule set occur infrequently compared with the occurrences of classifications a router must perform, it is unclear how often updates actually appear. On the one hand, in [101] Lakshman and Stiliadis assume that updates happen in intervals of tens of seconds. On the other hand, in [75] update times of tens of milli-seconds are a reasonable guess. And finally, in [145] it is shown that every single packet transmission may trigger the insertion of a classification rule.

Note that Feldmann and Muthukrishnan present a framework in [51] by which the general multi-dimensional packet classification problem can be mapped to several instances of a longest prefix match problem. That is, all algorithms presented in the preceding subsection can also be utilized to solve the classification problem.

Overview: The classification problem can be solved by several search approaches which are implemented by bitmap intersection [101], walking through so-called fat inverted segment trees [51], or heap on trie data structures [76]. These algorithms provide different – sometimes configurable – trade-offs between search and update times as well as the storage requirements. Since they employ general data structures, their worst-case behavior does not depend on the actual range distribution. Opposed to that, the method of hierarchical cuttings [75] adaptively subdivides the range space into subproblems which can be solved by linear search. Finally, tuple space search [145] starts with a linear search in a rule set, whose number of rules has been reduced by heuristics, and then jumps into a hash table.

- ***Linear search:*** The simplest algorithm one may imagine in order to find the best matching classification rule is a linear search over all ranges and rules which are compared by decreasing priority value. However, the search time as well as the storage requirements increase linearly with the number of rules N and dimensions d making this search algorithm only feasible for a small set of rules and dimensions. At least, the update time can be bounded by $O(\log N)$ if binary search is applied to a sorted data structure which is arranged by decreasing priority.

- ***Bitmap intersection:*** The optimization of one goal must usually be traded off for another goal. Fast search times can be achieved by precomputing complex data structures at the expense of larger update times. In [101], the set of possibly overlapping ranges is subdivided into non-overlapping intervals for each dimension. The set of rules in which the interval is a part of the corresponding range is assigned to each interval. Rules within a set are sorted by priority and the sets are stored as bitmap vectors. The classification task can thus be reduced to an intersection of sets implemented by a logical AND operation among bitmaps. These bitmaps represent the results of individual interval searches in each dimension using binary search, for instance. The highest priority entry in the resulting bit vector then corresponds to the best-matching rule. The search complexity reduces to $O(d \log N)$, the update complexity increases to $O(dN)$, and the storage requirements to $O(dN^2)$, respectively. Further refinements of

the scheme reduce the storage space requirements by coding only the difference between bitmaps at the cost of a higher number of memory accesses.

- ***FIS tree search:*** The so-called fat and inverted segment (FIS) tree presented in [51] also heavily employs precomputation to speed up search times. A FIS tree is a balanced, inverted t-ary tree with an arbitrary number of levels l. Thus, with $t = \lceil N^{\frac{1}{l}} \rceil$, each node has a pointer to its parent and at most t incoming arcs. Leaves represent the non-overlapping intervals of the range space defined by the end points of the ranges in the rule set, looking at a single dimension. Internal nodes denote the union of intervals stored at their child nodes. A rule will be assigned to a node if the node's interval is part of the corresponding range and if the parent node's interval is not part of the range. In a recursive manner, further FIS trees are constructed at every node projecting the set of rules which has been assigned to a node to the next dimension. FIS trees for the last dimension only have one level. In this way, the space required can be bounded by $O(N(lN^{\frac{1}{l}} \log N)^{d-1})$ and the search time by $O(l^{d-1} \log N)$. By varying the number of levels l in a FIS tree space requirements and search time behavior can be traded off. The support for dynamic updates of the data structure needs further modifications of the FIS tree. The update time for the multi-dimensional case has not been bounded in [51]. In the one-dimensional case, updates show a complexity of $O(lN^{\frac{1}{l}} \log N)$.

- ***Heap on trie:*** In [76], a range is split into a set of maximal prefixes. In the IPv4 case, for instance, there are 62 such prefixes. These prefixes are organized in a binary trie [96] data structure (basically a binary tree where the sequence of bits of the search argument is used to select a branch in each level of the tree to navigate through the tree). A range is assigned to the trie nodes which represent the range's set of maximal prefixes. Since ranges may overlap, several ranges can be assigned to a trie node. Therefore, ranges associated with a particular trie node are arranged in a heap which is ordered by cost or priority values. In the multi-dimensional case, a hierarchical trie is used, one level per dimension except the last one, for which the described heap-on-trie data structure is built. The space consumption can be bounded by $O(NW^d)$, the search time by $O(W^d)$, and the update time by $O(W^d \log N)$ operations where W is the maximal number of bits used to represent a range in one dimension. A second data structure is proposed that reduces the update time to $O(W^{d-1} \log N)$ at the expense of possibly larger search times which are bounded by $O(W^d \log N)$ operations.

- ***Hierarchical cuttings:*** Precomputation combined with heuristics to take advantage of the characteristics of real-life rule sets is used in [75] where a decision tree data structure is traversed to find a suitable leaf node. A leaf stores a set of rules that is searched linearly. During the building of the data structure, the range space is cut into a variable number of pieces. Each decision only affects the division along a single dimension. The cutting is performed recursively until the number of rules associated with a single piece of the range space falls be-

low a defined threshold. These pieces become leaves of the decision tree. The number of intervals per cutting, the choice of the dimension to cut along, and the threshold to stop the process of subdividing are all free parameters of the algorithm. Although heuristics work well with current rule sets, it is inherently clear that the algorithm may perform as bad as linear search in the worst-case.

- ***Tuple space search:*** The approach presented in [145] also combines precomputation and heuristics. The data structures are derived by the observation that only a small number of combinations of prefix and range lengths are used in current rule sets. Hence, filter rules with the same prefix and range lengths in every dimension are represented by a d-tuple of length values. The tuple set is searched linearly for a match. The corresponding significant bits of a packet header are then used as a key for an underlying hash table. One may think of two extreme cases: on the one hand, rules could not be compressed into tuples at all and a linear search of rules would actually be performed. On the other hand, all filter rules could be matched by a single tuple and packets would be classified by hashing with no need of a linear tuple search. Therefore, the storage and update time requirements are determined for the most part by the choice of the hashing function and the hash data structure. Again, in the worst-case, the search time can be as bad as in the case of a linear search since tuple space search tries to exploit the structure of existing rule sets.

2.1.3 Policing

Problem statement: After having classified a packet its context information is available. In particular, the traffic flow to which the packet belongs has been determined. Service guarantees can now be checked by verifying Service Level Agreements (SLAs) between customers and the provider of a service for that flow. This is done by measuring the flow's actual traffic profile. If the current packet is within the guaranteed profile, the packet will be processed without any restriction and marked as conforming to the SLA. If the profile is not kept, there are different options which depend on the currently available resources. Packets that violate the SLA could be immediately dropped. If, however, sufficient shared resources are available, the packet can be marked as non-conforming and nevertheless be processed at a somewhat degraded service level.

In addition, a packet may be delayed before policing in order to shape the flow according to a profile. This approach will be advantageous if there is some background knowledge that the flow has actually entered the network according to that profile and has been reshaped by the characteristics of intermediate network nodes. If the shaper and the policer use the same profile for the flow, the shaper may take over the marking task. Without prior knowledge about the expected profile, shaping does not make much sense because the shaper could always run into congestion and drop packets. In this case, a more sensible solution would be to spend more memory which is controlled by the queue manager at the link scheduler rather than buffer space for shapers. If no SLA exists for a given flow, a policer may nevertheless be employed to bound the amount of

unspecified traffic.

In the following, it is shown how traffic profiles are usually specified. Then, mechanisms to meter these profiles are described. If these measurement blocks are additionally allowed to delay packets, they can also be used for shaping traffic. The concept of arrival curves α is used which will formally be introduced in Section 2.2 to describe a worst-case traffic envelope for a given flow. An arrival curve $\alpha(t)$ bounds traffic for any interval of length t. That is, if the lengths of packets passing a defined place in the network are monitored, the sum of the packet lengths within any measuring interval of length t will not be more than $\alpha(t)$.

2.1.3.1 Traffic specifications

- (σ, ρ) ***model:*** With the (σ, ρ) model, the maximum burst size σ (an amount of bits) and the long-term bounding rate ρ of a traffic source are specified. The traffic can then be bounded by the arrival curve $\alpha(t) = \sigma + \rho t$.

- ***TSpec:*** The TSpec [141] has been introduced by the Internet community to treat QoS reservations and can be seen as a conjunction of two (σ, ρ) specifications. A TSpec is defined by a peak rate p, an average rate r, a burstiness b, and the maximum packet size M. The TSpec specifies one further parameter, the minimal policed unit m which determines that packets smaller than m should be treated as packets being of length m. In the end, traffic bounded by a TSpec is described by an arrival curve $\alpha(t) = min\{M + pt, b + rt\}$ with $M \geq m$, see Fig. 4.

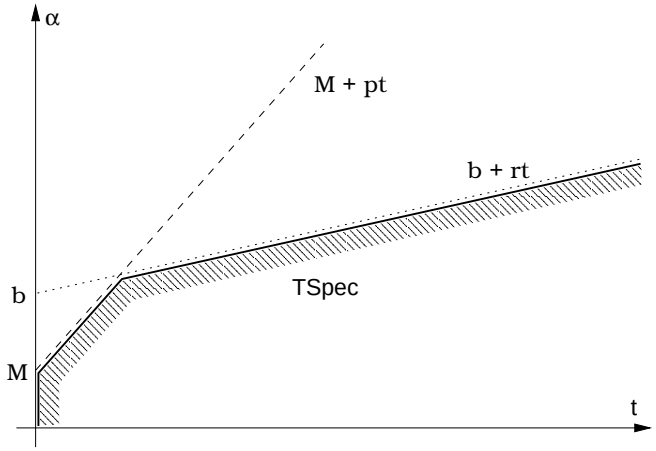

Fig. 4: The arrival curve for a *TSpec* traffic specification.

- $(\mathbf{X_{min}}, \mathbf{X_{ave}}, \mathbf{I}, \mathbf{S_{max}})$ ***model:*** This model has been introduced in a framework for real-time communication over packet-switched networks [8]. X_{min} denotes the minimal inter-arrival time of packets, X_{ave} the minimal average inter-arrival time in any interval of length I, and S_{max} the maximal packet size respectively.

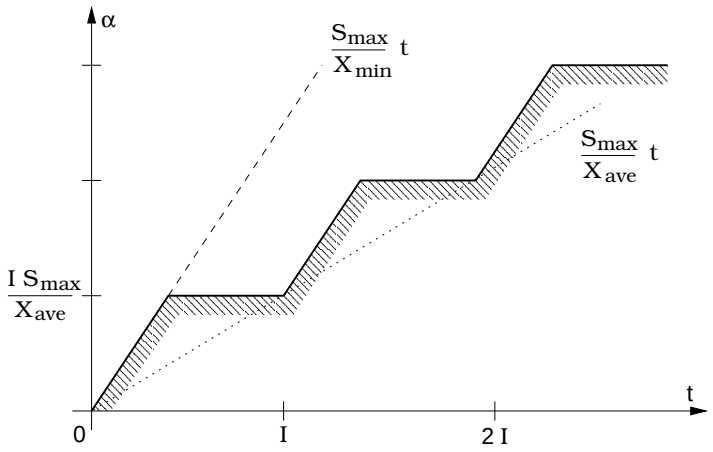

Fig. 5: The arrival curve for a $(X_{min}, X_{ave}, I, S_{max})$ traffic specification.

The corresponding arrival curve has been determined in [165] to be (see Fig. 5) $\alpha(t) = \lfloor \frac{t}{I} \rfloor \cdot \frac{I \cdot S_{max}}{X_{ave}} + \min\{\lceil (\frac{t}{I} - \lfloor \frac{t}{I} \rfloor) \cdot \frac{I}{X_{min}} \rceil, \frac{I}{X_{ave}}\} \cdot S_{max}$.

2.1.3.2 Metering

- ***Token bucket:*** A token bucket is defined by two parameters: its capacity B and the fill rate R. The bucket is continuously filled by the rate R with tokens representing units of Bytes up to the level B. The bucket is initially filled up with tokens. Traffic is allowed to pass the token bucket in the presence of a sufficient amount of tokens. The corresponding amount of tokens representing the length of the current packet is then taken from the bucket and the packet is marked as conforming to the profile. Otherwise, no tokens are taken from the bucket and the packet is marked as non-conforming. In this way, the traffic is allowed to have a certain burstiness up to a size of B. However, since the bucket is refilled by the rate R, the traffic is bounded in the long term by R. That is, conforming packets can be described by a (σ, ρ) model with $\sigma = B$ and $\rho = R$. ATM's generic cell rate algorithm (GCRA [7]) is a metering algorithm that is based on a token bucket. The traffic is checked whether it is within the bounds of a peak rate (PCR) and the cell delay variation tolerance ($CDVT$). The GCRA is thus equivalent to a token bucket with rate $R = PCR$ and capacity $B = CDVT \cdot PCR + \mathrm{SizeOf}(ATMCell)$.

- ***Conjunction of token buckets:*** Accordingly, traffic may be marked as conforming to a TSpec by checking two token buckets running in parallel. One bucket has the size M and is filled by the peak rate p, the other one has the capacity b and is filled by the average rate r. A packet will be allowed to pass the dual token bucket as conforming to the TSpec if there is a sufficient amount of tokens in each of the buckets. Then, the corresponding amount of tokens is taken from both buckets.

- ***Nested token buckets:*** In order to enable a graceful service degradation if traffic does not meet a certain profile but is within a somewhat less restrictive profile,

one could think of nesting token buckets. In its simplest form [81] by nesting two token buckets with profiles (b, r) and (B, R), $b \leq B$ and $r \leq R$, a packet will be marked as conforming to the (b, r) profile if there are enough tokens in both buckets. Then, tokens are taken from both buckets and the packet is marked for premium service. If, however, the packet does not fit into the (b, r) but only into the (B, R) profile, the packet is marked for degraded service and tokens are only taken from the (B, R) bucket. At last, if there are not enough tokens in the (B, R) bucket, the packet will be marked as non-conforming and no tokens will be taken from any bucket.

- ***Testing a*** $(\mathbf{X_{min}}, \mathbf{X_{ave}}, \mathbf{I}, \mathbf{S_{max}})$ ***profile:*** The arrival time a_k of the kth packet of a traffic flow must be checked whether it is valid compared with the arrival times of preceding packets. That is,

$$a_k \geq \max\{a_{k-1} + X_{min}, a_{k-\lfloor \frac{I}{X_{ave}} \rfloor + 1} + I\} \text{ with } a_k = -I, k \leq 0$$

must be true in order to mark the current packet as conforming to the profile assuming a length of the packet less or equal S_{max}. Consequently, one must be aware of the traffic's history by storing the arrival times of the preceding $\lfloor \frac{I}{X_{ave}} \rfloor - 1$ packets which conformed to the profile.

2.1.3.3 Shaping

All the mechanisms described in the preceding subsection can be used as a basis for a corresponding shaper entity. The only difference is that a packet is never dropped – unless the shaper runs out of buffer space – but is delayed until it finally conforms to a given profile.

A special case of a shaper is the fluid model of a ***leaky bucket*** that is often confused with a token bucket. Network traffic is assumed to pour into a bucket with capacity B. Fluid, i.e. network traffic, is continuously leaking out of the bucket through a hole in the bottom at a constant bit rate cbr as long as there is fluid in the bucket. Fluid will be lost if the bucket overflows. Thus, a leaky bucket may tolerate a certain level of burstiness of the incoming traffic at its input until the maximum level of the bucket is reached but will always generate constant bit rate traffic at its output as long as there is backlog in the bucket. Opposed to that, a token bucket allows a bounded burstiness at its output because traffic is immediately forwarded as long as there are enough tokens in the bucket. The token bucket's level however regenerates with constant bit rate.

With a slight modification, a leaky bucket can be used as a smoothing element by introducing some additional delay. If it is known that a constant bit rate source with rate cbr has been partially delayed in the network and now has a jitter of δ, the constant bit rate flow can be reconstructed without any losses and any gaps by feeding a leaky bucket of size $\delta \cdot cbr$ with it. At the beginning, one must wait until the buffer is half-filled and then start to read out the bucket at the constant bit rate cbr.

2.1.4 Queuing

Problem statement: After a packet has been admitted for a possible transmission, it must be buffered in the system until it will be either chosen by the link scheduler for transmission or be discarded in case of a congested link. It is the responsibility of a queue manager to operate the packet storage space which may include dynamically allocating and deallocating memory to store or release packets as well as coping with congestion, i.e., choosing packets to discard.

Ideally, the behavior of the different flows should be isolated from each other. Packets marked as conforming to their profiles should always be stored regardless of other greedy traffic sources and surplus storage space should be shared in a fair manner. In order to balance the separation of flows and the number of flows that can be managed, different approaches are possible:

- ***Single queue:*** Packets from all flows are simply enqueued in the order they arrive at the queue manager without storing any addition per flow state. Although a high number of flows can be aggregated this way since one does not depend on any per flow state information, the search for a packet of a particular flow needs an exhaustive linear search among all enqueued packets.

- ***Separate queues:*** There is a separate FIFO queue for each flow. The memory space is statically subdivided into parts which are exclusively assigned to distinct flows. The expense for organizing a FIFO queue can be kept at a moderate level, e.g. by employing ring queue data structures based on arrays. While this approach achieves perfect isolation of the flows, storage resources may be wasted since flows are not allowed to exploit memory which is currently unused by another flow.

- ***Shared memory:*** FIFO queues for the flows are organized as linked lists of data segments of a decent size. Thus, the contents of a packet may be distributed over several segments. Memory space is dynamically allocated and released for every single packet, e.g. by maintaining a list of free segments. By defining individual thresholds of memory space utilization for each flow according to some reservation rule, a flow can be protected against other flows as long as its occupation of the memory is below its threshold. Memory beyond these thresholds can be shared arbitrarily at the expense of the preservation of more complex data structures and per flow state information.

Independently of the organization of the memory the congestion behavior of the queue manager can be determined using the following guidelines:

- ***Congestion avoidance:*** A congestion situation may be avoided by preventively discarding packets dependent on the system's state before the appearance of congestion.

- ***Congestion recovery:*** Packets are dropped in times of congestion to avoid deadlocks.

In order to succeed, the queue manager may apply

- ***Rejection of packets at the arrival:*** Packets may not be allowed to enter the queue manager dependent on the system's state.
- ***Pushing out already stored packets:*** Packets that have already been enqueued in the queue manager are discarded at the arrival of new packets. Common actions are to drop packets from the front or from the tail of a queue as well as to randomly select a packet [21].

Commonly used implementations of *congestion avoidance* approaches are:

- ***Early Packet Discard (EPD):*** Incoming packets will be immediately dropped by EPD [132] if the current size of the queue passes a fixed threshold.
- ***Random Early Detection (RED):*** RED [54] starts to drop incoming packets with a defined probability as soon as the average size of allocated memory exceeds a given threshold. The probability to be dropped then increases linearly with the memory size. If a second, larger threshold is passed, every incoming packet will be dropped. Note that the calculation of the average memory size introduces further parameters. Flow random early drop (FRED [105]) is an extension to RED that employs per flow state information – in particular queue lengths for individual flows – in order to more fairly drop packets from flows depending on individual memory utilizations.

Congestion recovery mechanisms – partly extended by using policing information and congestion avoidance ideas – have been implemented in:

- ***Longest Queue Drop (LQD):*** LQD [156] simply pushes out packets from the currently longest flow queue. Assuming an equal reservation of resources for all flows, flows are protected against misbehaving flows that show a longer backlog of packets and thus experience higher loss rates. However, one must keep track of the longest queue in the system.
- ***Extended Threshold Policy (ETP):*** ETP [34] pushes out packets in order to cope with congestion. Congestion may be avoided because all packets which have been marked as non-conforming by the policer and which are currently stored above a defined threshold will be discarded if a packet marked as conforming arrives. Note that a packet arrival may initiate several packet discards.
- ***Extended Simulated Protective Policy (ESPP):*** ESPP [34] also pushes packets out in order to recover from congestion. A second queuing system is maintained as a reference which only deals with conforming traffic. The goal is to ensure that the main system offers the same service for conforming traffic as the reference system at any point of time. That means in particular, the main system should always offer at least as much buffer space to conforming traffic as the reference system. Therefore, incoming packets which conform to their profile will push out non-conforming packets if no free buffers are available.

2.1.5 Link scheduling

Problem statement: A link scheduler is a kind of arbiter that must decide which of the buffered packets will be transferred next through an outgoing link of a networking node. The scheduler may use further information such as the system state, service level agreements, or recent traffic characteristics metered by a policer to guide and support its decision which packet to choose next.

There are several features by which schedulers may be distinguished:

- ***Fairness:*** Schedulers will be considered to be fair if surplus bandwidth is distributed to backlogged flows in proportion to their reservation, i.e., they should not give preference to any flow.
- ***Efficiency:*** The complexity of an implementation is valued here. Issues like whether and how the number of operations and memory accesses depends on the number of flows and packets in the system are answered.
- ***Worst-case behavior:*** The service a flow receives according to an SLA should not depend on any properties of other flows, i.e., the behavior of flows should be isolated from each other. Thus, conforming flows should not be disturbed by greedy flows.
- ***Quality of service guarantees:*** Some schedulers may only offer some distinct rates and not arbitrary reservations and provide worse delay bounds than others. Yet other ones are able to decouple response time and throughput guarantees to some extent. In Section 2.2, it is discussed in detail how service guarantees have an effect on resource requirements and depend on traffic specifications.
- ***Utilization:*** The number and variety of flows that a scheduler is able to permit transmission according to their SLAs – in particular with respect to guaranteed delay bounds – may vary from scheduler to scheduler. The utilization of a scheduler is often determined by its schedulability region.

2.1.5.1 Static priority-driven schedulers

- ***First Come First Served (FCFS):*** An FCFS server only provides a single priority level. Therefore, packets are served in the order of their arrival. An underlying queue manager can easily be implemented since only a single queue must be maintained. The overall complexity is also very low: insertion and deletion from the queue as well as a schedule decision can be done in $O(1)$ time. Obviously, FCFS does not provide any isolation of flows or any fairness since a greedy flow may capture an arbitrary fraction of the link bandwidth. Moreover, individual delay guarantees are as bad as for a single flow occupying the whole queue: in the worst-case, assuming that an incoming packet has been stored in a nearly filled-up queue, the packet must wait for service until all the preceding packets in the queue have been processed. Although these disadvantageous properties of FCFS including the possible collapse of whole

networks are known for a long time [116], FCFS combined with a RED queue manager, for instance, is still a very popular solution for backbone routers due to its simplicity. Hence, FCFS is still discussed in recent publications [72, 156]. The suggestion is to combine FCFS with queue managers that retain per-flow states so as to enhance FCFS with some isolation properties. These approaches accept bad delay characteristics and a high overhead in terms of memory space to achieve a somewhat passable schedulability region – necessary schedulability tests for (σ, ρ) constrained sources in terms of queuing resources and rate reservations are derived in [72] – in favor of an implementation of low complexity. This may still be a reasonable choice for high-speed backbone routers.

- ***Static Priority (SP):*** A natural extension to FCFS is to provide a bounded number of distinct FIFO-organized queues and associate a fixed priority level with each queue. A packet of low priority will only be transmitted if all higher priority queues are empty. In this way, a decision of the scheduler on the next packet to transmit remains a task of complexity $O(1)$. However, it is still possible that a greedy flow of a particular priority starves all other flows of the same and of lower priority. That is, high priority flows are only protected against misbehaving flows of lower priority. Moreover, high priority flows benefit alone from surplus bandwidth as long as they are backlogged. Necessary and sufficient schedulability tests for flows constrained by concave arrival curves are derived in [40] in terms of arbitrary delay bounds – a single delay bound per priority level. Sufficient tests for flows bounded by the non-concave $(X_{min}, X_{ave}, I, S_{max})$ model are presented in [170, 171]. Finally, necessary and sufficient conditions for arbitrary arrival curves can be found in [104]. The conditions are not discussed any further at this point since they are significantly more complex than the schedulability tests for EDF-based schedulers. Since EDF is also the more flexible approach compared with SP, the conditions for EDF will be considered in more detail later in this subsection.

- ***Round-Robin (RR) based schedulers:*** In order to bound the time interval by which flows of a high priority level may starve lower prioritized flows, one could think of limiting the amount of service a particular priority level receives in a SP-based system at a point of time. One could proceed to the next lower priority level after the current priority level has taken its quantum. After having served the lowest priority level, one would begin again with the queue of highest priority. Levels without backlog would be skipped. This way, a share of the link bandwidth can be guaranteed to each priority level even under heavy load and surplus bandwidth is also shared in a fair manner according to quantum values. Different priority levels can be considered by individually adjusting the quantum value of each priority level. The scheduling decision remains a constant time operation. However, the delay properties still depend on the number of distinct queues that are currently active in the system. The Round-Robin idea applied to networking nodes has been introduced in [116, 117, 78] using fixed-sized quantum values. Hierarchical Round-Robin (HRR) [94] allows variable-sized quantum values and several levels of HRR schedulers to take dif-

ferent packet sizes and rate reservations into account as well as schedule flows within a particular priority level fairly by RR instead of FCFS. Finally, Deficit Round-Robin (DRR) [142] allows to accumulate the shares of a quantum that have not been exploited during preceding scheduling rounds as long as the assigned queue is backlogged.

2.1.5.2 Dynamic priority-driven schedulers

Dynamically generated priorities are used for schedulers of this class. That is, since the priority of a packet is derived during the run-time of the system, packets must be dynamically sorted according to these priority values. The scheduler cannot apply any fixed scheme as described in the preceding subsection but serves the packet with the currently highest priority from a sorted queue. The sorted data structure is also often called a *priority queue*. That means, not only the calculation of priority values is needed, but also a sorted data structure must be maintained. However, it is sufficient to sort only the N head-of-queue elements from each of the N flow queues. Thus, N distinct FIFO queues can be kept up. Additionally, for most of the following algorithms it is sufficient to compute a priority value when a packet reaches the head of a FIFO queue and not already on arrival at the system. Data structures for priority queues are evaluated in general in [133, 93, 96, 158], applied to packet scheduling algorithms in [95, 148], and hardware building blocks for network applications are described in [30, 130, 112].

- ***Weighted Fair Queueing (WFQ):*** The approach to adapt the behavior of a perfectly fair fluid server to the time-multiplex in packet networks is the basis for a variety of packet scheduling algorithms. The idea is introduced in [44] under the name *Fair Queueing* (FQ) and is thoroughly analysed in [122]. A fluid server is configured by N positive real numbers (weights) $\Phi_1, \Phi_2, \ldots, \Phi_N$ that are assigned to N distinct FIFO queues. During any time interval $(\tau_1, \tau_2]$ when there are exactly $n \in [1, \ldots, N]$ queues backlogged the fluid server serves the n packets at the head of the corresponding queues simultaneously, each at a rate

$$r_n(t) = \frac{\Phi_n}{\sum_{j \in B(\tau_2)} \Phi_j} \cdot R(t), t \in (\tau_1, \tau_2] \tag{2.1}$$

where $B(\tau_2)$ is the set of backlogged queues which is constant in the time period $(\tau_1, \tau_2]$ and $R(t)$ is the – possibly variable – link speed respectively. Therefore, let $W_n(\tau_1, \tau_2)$ be the amount of traffic from queue n served in interval $(\tau_1, \tau_2]$[2],

$$\frac{W_n(\tau_1, \tau_2)}{W_j(\tau_1, \tau_2)} \geq \frac{\Phi_n}{\Phi_j}, j = 1, 2, \ldots, N$$

is true for any queue n that is continuously backlogged in the interval $(\tau_1, \tau_2]$. In other words, the rate $r_n(t) = \frac{\Phi_n}{\sum_{j=1}^{N} \Phi_j} \cdot R(t)$ can be guaranteed to queue n by the

[2] $W_n(\tau_1, \tau_2)$ includes service provided after time τ_1 to packets of queue n whose transmission started before time τ_1 and the service provided until time τ_2 to packets whose transmission is finished after time τ_2.

fluid server and surplus bandwidth is shared fairly in proportion to the weights Φ_n of the currently backlogged queues. Without restricting the generality of the method, assuming normalized weights $\Phi_n \in (0, \ldots, 1]$ and a stable system, i.e., $\sum_{j=1}^{N} \Phi_j \leq 1$, the worst-case rate guarantee can be uncoupled from the weights of all other flows and one can state that a fluid server guarantees a rate $r_n(t) = \Phi_n \cdot R(t)$ to queue n in the worst-case independently of the behavior of all other queues.

Since packets cannot be served simultaneously in a packet system on a single outgoing link, the packetized version of a fluid server – the Weighted Fair Queueing (WFQ) server – schedules packets according to the order in which they would finish service in the fluid system. Hence, a WFQ server must emulate the fluid system in the background in order to function properly. Before the actual WFQ algorithm can be presented, a method to keep track of the fluid server is introduced which is based on the evolution of a single virtual time measure.

The evolution of virtual service in the emulated fluid system: By restricting the outgoing link to a constant bit rate, the rate allocation in eq. (2.1) may only change at packet arrivals and departures in the fluid system because the set of backlogged queues $B(\tau_2)$ may only alter at these events. In [122], an algorithm derived from the concept of *virtual time* – often also called virtual service – is presented that keeps track of the fluid server.

– *Virtual time evolution:* The virtual time $V(t)$[3] is defined to be zero when the server is idle. For any interval dt with a constant set of backlogged flows within any busy period, the virtual time evolves as follows:

$$V(t + dt) = V(t) + \frac{dt}{\sum_{j \in B(t)} \Phi_j} \cdot R \tag{2.2}$$

and each backlogged queue receives service $r_n(t) \cdot dt = (V(t+dt) - V(t)) \cdot \Phi_n$ in interval dt. In this way, one only needs to keep track of a single service measure $V(t)$ in order to derive individual service amounts for each queue which are needed to fairly schedule packets from different queues.

– *Scheduling tag calculation:* At each packet arrival, the service levels S and F of the virtual service $V(t)$ are computed at which the packet would start to receive service (S) and finish service (F) in the fluid system. Suppose that the kth packet of queue n arrives at time $a_{n,k}$ and has the length l_n^k. The start and finish virtual service times can be computed by[4]

$$\begin{aligned} S_n^k &= max(F_n^{k-1}, V(a_{n,k}^-)), \text{ with } F_n^0 = 0 \\ F_n^k &= S_n^k + \frac{l_n^k}{\Phi_n} \end{aligned} \tag{2.3}$$

[3] By applying the notation of weights Φ_n the virtual time $V(t)$ is measured in units of $Bits$. An alternative notation can be derived by using the reserved rates r_n instead of Φ_n so that $V(t)$ is measured in $seconds$.

[4] In the following, the notation $\tau^{\{-|+\}}$ for any point of time τ of an event means just prior the event and just after the event at time τ respectively.

The finish times F of all packets enqueued in the fluid system are sorted in the order of increasing times. The maximum operation for the calculation of S_n^k in eq. (2.3) would prevent a queue from accumulating credits for unused service if the queue was not backlogged during a busy time of the scheduler.

– *Real time of next packet departure:* Let F_{min} be the minimal finish time of packets in the fluid system, the real time $next(t)$ can be calculated at which the next packet will leave the fluid system assuming that there will be no arrivals in the interval $(t, next(t)]$ by

$$next(t) = t + (F_{min} - V(t^-))\frac{\sum_{j \in B(t^+)} \Phi_j}{R} \tag{2.4}$$

That is, $next(t)$ is the next point of real time at which the set of backlogged flows may possibly be altered and the slope of the virtual time may change (eq. (2.2)).

The ***WFQ algorithm*** can now be described as follows:

Alg. 1: (WFQ with emulated fluid server)

System state parameters of the fluid server:		
R	*[Bit/s]*	*link rate*
N		*number of flows*
Φ_n, $1 \leq n \leq N$		*WFQ weights*
System state variables of the fluid server:		
$V(t)$	*[Bit]*	*virtual time*
$B(t)$		*set of backlogged flows*
F_n^{k-1}, $1 \leq n \leq N$	*[Bit]*	*virtual finish times of preceding packets*
F_{min}	*[Bit]*	*minimum virtual finish time among all enqueued packets*
$next(t)$	*[s]*	*point of real time of next packet departure*
Input parameters:		
$a_{n,k}$	*[s]*	*arrival time of the kth packet of flow n*
l_n^k	*[Bit]*	*length of the kth packet of flow n*

At each packet arrival *at time $t := a_{n,k}$*

1. *If the fluid system was idle just before the packet arrival, reset the system by setting the virtual time V and all preceding scheduling tags F_n^{k-1} of all flows to zero, else update $V(t)$ according to eq. (2.2).*
2. *Calculate the finish time for the packet with eq. (2.3).*
3. *Add n to the set of backlogged queues $B(t)$ if $n \notin B(t)$.*

4. *Sort the packet into two priority queues: one queue models the behavior of the fluid system and the other represents the actual packet system.*
5. *Calculate the real time of the next packet departure $next(t)$ in the fluid system with eq. (2.4).*

At each packet departure event *in the fluid system at time $t := next$*

1. *Update the virtual time $V(t)$ according to eq. (2.2).*
2. *Dequeue the packet with the smallest finish time from the priority queue of the fluid system. Let the corresponding flow identifier be n.*
3. *If the flow n is no longer backlogged in the fluid system, take n from the set of backlogged queues $B(t)$.*
4. *If there are still packets backlogged in the fluid system, calculate the time of the next event $next(t)$ in the fluid system with eq. (2.4).*

The packet system serves packets independently of the departure events in the fluid system. Packets are served in order of increasing finish times. It is shown in [122] that by applying WFQ the packet system can only be a packet length behind the service of the ideal fluid system in the worst-case. However, there is a tremendous overhead for the calculation of the scheduling tags because events may appear frequently in the fluid system since virtually all backlogged flows may finish service at the same time.

In order to find a suitable trade-off between the complexity of a scheduling algorithm, the fairness of the distribution of excess bandwidth, and the provision of sharp delay bounds, different approaches have been applied to implement WFQ.

– *Self-clocked fair schedulers:* So-called *self-clocked* methods no longer emulate a fluid system but estimate scheduling tags by the tags of packets which are currently queued in the packet system. *Self-Clocked Fair Queueing* (SCFQ) [62] uses the finish time of the packet currently in service for the estimation of the virtual time $V(t)$ of the fluid server in eq. (2.2). Contrary to that, *Start-time Fair Queuing* (SFQ) as described in [64] uses the start time of the packet currently in service for the estimation and serves packets in increasing order of their start times. *Minimum Starting-tag Fair Queueing* (MSFQ) [33] serves packets in increasing order of their finish times and the virtual time of the fluid server is estimated by the minimum of the start times of backlogged flows. A second priority queue is therefore required. The same approach has been independently published in [36] under the name *time-shift scheduling*. Self-clocked algorithms decrease the implementation complexity of WFQ since the state of the fluid system is simply modeled. However, they usually provide poor delay bounds which may depend on the number and the reservations of other flows passing the scheduler.

– *Approximation by potential functions:* More methodical approaches of fair queuing designs that also exploit the self-clocked idea are based on the theory of *Rate-Proportional Servers* (RPS) [152]. A scheduler of the type RPS keeps track of the state of the fluid system by a system potential function that models the behavior of the virtual time $V(t)$. A so-called base potential function is employed in order to recalibrate the system potential at certain points of time. The system potential then increases linearly between recalibrations – assuming the system is not idle – so as to resemble the increase of the virtual time $V(t)$ in eq. (2.2). Different base potential functions as well as different recalibration time intervals can be chosen in order to find a suitable trade-off between fairness and complexity of the scheduler. RPS-based schedulers achieve the same worst-case delay bounds as WFQ. *Starting Potential-based Fair Queueing* (SPFQ) and *Frame-based Fair Queueing* (FFQ) have been presented in [150]. SPFQ recalibrates the system potential at every packet departure. The base potential is updated at every packet arrival and is set to the minimum start potential of all backlogged flows, that is, the potential at which the corresponding packet would begin getting service in the fluid system. Thus, SPFQ is very similar to MSFQ. MSFQ however does not use a system potential and moreover recalibrates at packet arrivals. FFQ uses a simpler base potential than SPFQ and larger intervals between recalibrations at the expense of fairness. *Minimum Delay Self-Clocked Fair Queueing* (MD-SCFQ) [31] uses the same recalibration intervals as SPFQ together with a simplified base potential which does not need to maintain a second priority queue in order to manage start potentials. However, MD-SCFQ can achieve better fairness than SPFQ for certain flow settings.

– *Eligible packet selection:* Although the amount of service by which a WFQ system may be behind a fluid system is bounded, the WFQ system schedule can be quite ahead of the fluid system [122]. This behavior will show undesired properties if feedback congestion control is used, e.g. for the regulation of best-effort traffic [14]. In order to retain fairness between the flows sharing a link not only on the average but also on a fine time granularity, there are scheduling algorithms which use two distinct priority queues for sorting. *Worst-case fair Weighted Fair Queueing* (WF^{2}Q) [14] and its more efficient implementation WF^{2}Q+ [13] sort arriving packets according to their start time in the fluid system. Only packets which are eligible for transmission, that is, for which service would have been started in the fluid system, are then transferred to the second priority queue which is sorted according to finish times. WF^{2}Q+ does not need to emulate a fluid server but utilizes an approximation function similar to SPFQ and MSFQ. Opposed to these algorithms, WF^{2}Q+ considers only eligible packets for transmission. *Leap Forward Virtual Clock* (LFVC) [155] transfers packets from backlogged but oversubscribed flows to a second priority queue. Packets residing in this queue are not eligible for transmission yet. Care is taken that packets are written back to the first priority queue before any delay bound may be missed. WF^{2}Q and LFVC have in common that the full contents of one priority queue must pos-

sibly be copied to the other priority queue between two scheduling decisions in the worst-case.

– *Round-Robin variants:* There are scheduling algorithms with low complexity which enhance the concept of a Round-Robin scheduler with virtual service ideas. However, *Virtual Time-based Round-Robin* (VTRR) [32] cannot provide such sharp delay or fairness bounds as schedulers which use a fluid server as reference model. Note that *Deficit Round-Robin* (DRR) [142] is often used as a somewhat standardized comparison basis for fair schedulers and is thus usually considered to be a WFQ implementation of low complexity and accordingly degraded service guarantees.

– *Hierarchical grouping:* If a single level of flows or flow classes is not detailed enough so as to distinguish QoS, one may think about using several levels of schedulers within a hierarchy [13, 55, 153]. However, the delay and fairness properties of the schedulers are accumulated through the levels of the hierarchy. Moreover, looking at the latter paper ([153]) where a scheduler based on service curves is used [137], one should be aware of the complexity overhead involved if service curves were applied which model more complex behavior than sources constrained by token buckets.

Two measures have evolved in order to assess the fairness of an algorithm. In the fluid system, from eq. (2.1), it immediately follows that for any two queues i, j that are continuously backlogged in the interval $(\tau_1, \tau_2]$ and have guaranteed service rates r_i and r_j respectively, the following holds:

$$\left| \frac{W_i(\tau_1, \tau_2)}{r_i} - \frac{W_j(\tau_1, \tau_2)}{r_j} \right| = 0$$

Def. 7: **(Fairness index $\mathcal{F}$ by Golestani [62])** *Given any two queues i, j that are continuously backlogged in the interval $(\tau_1, \tau_2]$ and have guaranteed service rates r_i and r_j respectively, Golestani defines a fairness index $\mathcal{F}_{i,j}$ for the packet system by*

$$\left| \frac{W_i(\tau_1, \tau_2)}{r_i} - \frac{W_j(\tau_1, \tau_2)}{r_j} \right| \leq \mathcal{F}_{i,j} \tag{2.5}$$

That is, any two queues i, j that are continuously backlogged in any interval $(\tau_1, \tau_2]$ must not receive normalized service which differs from the other queue's service by more than $\mathcal{F}_{i,j}$.

Since in the packet system a packet transmission cannot be preempted, there is a lower bound for the fairness index given by $\mathcal{F}_{i,j} \geq \frac{1}{2}(\frac{L_i}{r_i} + \frac{L_j}{r_j})$ where $L_{\{i,j\}}$ are the maximum packet lengths of the corresponding queues.

Def. 8: **(Time Worst-case fairness index $\mathcal{A}$ by Bennett et al. [14])** *A* Time Worst-case Fair Index *(T-WFI) $\mathcal{A}_n$ is defined for a constant bit rate server which states that, for any time $a_{n,k}$ the kth packet arrives for queue n with a guaranteed service*

rate r_n, the delay of the packet should be bounded by the level of queue n at time $a_{n,k}$ and a parameter $\mathcal{A}_n$ by

$$d_{n,k} - a_{n,k} \leq \frac{Q_n(a_{n,k}^+)}{r_n} + \mathcal{A}_n \tag{2.6}$$

where $d_{n,k}$ specifies the time at which the packet departs the system – the last bit of the packet has been served – and $Q_n(a_{n,k}^+)$ is the length of queue n measured in Bits just after time $a_{n,k}$, i.e. including packet k.

That means, if the value of $\mathcal{A}_n$ and therefore the maximum delay experienced by a packet in queue n only depends on the state of queue n and parameters of the server such as the link rate, the server will be considered to handle queue n fairly because the delay properties of queue n are not influenced by other misbehaving flows. The T-WFI is generalized in [13] for variable rate servers and arbitrary points of time as in the following definition.

Def. 9: **(Bit Worst-case fairness index γ by Bennett et al. [13])** *The* Bit Worst-case Fair Index *(B-WFI) γ_n is guaranteed for flow n by a server if for any packet departing the system at time $d_{n,k}$ and for any interval $[\tau_1, d_{n,k}]$ during which queue n is continuously backlogged the following holds*

$$W_n(\tau_1, d_{n,k}) \geq \frac{\Phi_n}{\sum_{i=1}^{N} \Phi_i} W(\tau_1, d_{n,k}) - \gamma_n \tag{2.7}$$

where $W(\tau_1, d_{n,k})$ is the overall amount of traffic served in interval $(\tau_1, d_{n,k}]$. That is, each backlogged queue should get at least its service share $\frac{\Phi_n}{\sum \Phi_i}$ guaranteed by the server minus γ_n.

In other words, if the outgoing traffic W of the server was metered in an interval beginning at an arbitrary point of time τ_1 and ending at the time of the finished transmission of a packet from queue n, at least $\frac{\Phi_n}{\sum \Phi_i} W - \gamma_n$ Bits should belong to flow n assuming that queue n has continuously been backlogged during the metering interval.

The fairness index by Golestani looks at the normalized service of two queues whereas Bennett's fair index compares the link service with a queue's service. If the server is allowed to be one of the queues in Golestani's definition in eq. (2.5), one will see that Bennett's fair index in eq. (2.7) can be expressed as a subcase of Golestani's fairness index.

In Tab. 1 latency and fairness values for some selected scheduling algorithms are gathered from [62, 150, 31, 149]. The latency Θ_i is defined to be the worst-case latency that a maximum-sized packet of a beforehand idle flow i with guaranteed service rate r_i will experience if it arrives at an empty queue. Note that the T-WFI fairness index $\mathcal{A}_i$ in eq. (2.6) can be derived from Θ_i by $\mathcal{A}_i = \Theta_i - \frac{L_i}{r_i}$, where L_i is the maximum packet length of flow i. Eq. (2.6) can then be used

Tab. 1: Latency and fairness properties of selected WFQ-influenced schedulers. There are N flows. L_i is the maximum packet size of flow i and $L = \max_{1 \le n \le N} L_i$. The rate r_i is guaranteed to flow i by the scheduler that utilizes a link rate R. DRR is configured by assigning quantum values q_i to the flows.

Server	*Latency* Θ_i	*Fairness* $\mathcal{F}_{i,j}$, eq. (2.5)
WFQ	$\frac{L_i}{r_i} + \frac{L}{R}$	$\max(\frac{L_i}{r_i} + \frac{L}{r_j} + f_i, \frac{L_j}{r_j} + \frac{L}{r_i} + f_j)$ where $f_i = \min((N-1)\frac{L}{r_i}, \max_{1 \le n \le N} \frac{L_n}{r_n})$
SCFQ	$\frac{L_i}{r_i} + (N-1)\frac{L}{R}$	$\frac{L_i}{r_i} + \frac{L_j}{r_j}$
SPFQ	$\frac{L_i}{r_i} + \frac{L}{R}$	$\max(\frac{L_i}{r_i}, \frac{L_j}{r_j}) + \max_{1 \le n \le N} \frac{L_n}{r_n} + \frac{L}{R}$
MD-SCFQ	$\frac{L_i}{r_i} + \frac{L}{R}$	$\max(f_{i,j}, f_{j,i})$ where $f_{i,j} = \frac{L_i}{r_i} + \max\left(\frac{L}{r_j}, \max_{1 \le n \le N} \frac{L_n}{r_n} - \frac{r_i}{R - r_j}(\max_{1 \le n \le N} \frac{L_n}{r_n} - \frac{L_i}{r_i}) - \frac{L_i}{R}\right)$
DRR	$\frac{3 \cdot \sum_{n=1}^{N} q_n - 2q_i}{R}$	$\frac{3 \cdot \sum_{n=1}^{N} q_n}{R}$

to deduce schedulability tests by bounding the maximum level of flow queues. However, a more generic approach that not only looks at the available resources but also at the traffic characteristics will be shown in Subsection 2.2.

An undesired property of Self-Clocked Fair Queueing (SCFQ) and Deficit Round-Robin (DRR) is their dependence on the number of flows sharing the link when individual guarantees should be given. However, SCFQ achieves the best fairness of all algorithms. Assuming the same maximal packet length for all flows, MD-SCFQ realizes a better worst-case fairness than SPFQ. SPFQ in turn shows better fairness than WFQ. Since SPFQ and MD-SCFQ both belong to the class of RPS servers they have the same worst-case latency as WFQ. Obviously, one cannot have both good latency and fairness bounds. Moreover, one should always consider the implementation complexity in addition. WFQ must emulate the fluid system, SPFQ requires a second priority queue, and finally MD-SCFQ has to compute a somewhat complex base potential function.

- ***Earliest Deadline First (EDF):*** An exhaustive survey of the EDF scheduling discipline can be found in [147]. EDF applied to link scheduling means that each incoming packet from a traffic stream is assigned a deadline. Packets are served in order of increasing deadlines. The scheduler sets a packet's scheduling tag to the deadline at which it will be sent at the latest if it arrives according to a traffic specification. That is, a deadline is calculated by adding the worst-case delay bound – the guaranteed delay – to the expected arrival time of the packet.

A packet may be delayed longer than its local delay bound if its actual arrival time is smaller than the expected arrival time. Since traffic streams are seldom periodic but rather sporadic, the most common EDF-based packet schedulers [160, 53] use the $(X_{min}, X_{ave}, I, S_{max})$ traffic profile as a basis for deadline calculations. That means, characteristics of the outgoing traffic such as the average rate are inherently determined by the traffic profile of the incoming traffic whereas the worst-case delay bounds are guaranteed by the scheduler. The allocation of both – the profile and the delay bound – can arbitrarily be chosen to some extent, i.e., flows with a low average rate may reserve sharp delay bounds as long as the system remains schedulable. Opposed to that, WFQ-based schedulers cannot weigh rate against delay guarantees since a WFQ weight always corresponds to a fixed share of the available link bandwidth. This is why EDF allows larger schedulability regions [57]. The necessary and sufficient schedulability condition for flows specified by arrival curves is derived in [104]. A slightly less restrictive and sufficient condition is stated in [58] that is presented here due to its simple graphical interpretation.

Theor. 1: **(EDF schedulability test by Georgiadis et al. [58])** *A set of N flows given by their arrival curves $\alpha_n(t)$ and assigned deadlines d_n is* EDF-schedulable *if:*

$$R \cdot t - L \geq \sum_{n=1}^{N} \alpha_n(t - d_n), \qquad t \geq 0 \tag{2.8}$$

where $L = \max_{1 \leq n \leq N} L_n$ is the maximum packet length of all flows and R is the link rate.

Ex. 3: **(EDF schedulability test with TSpec-compliant flows)** *In Fig. 6 an example for the EDF schedulability test is shown for two flows that are specified by TSpecs with arrival curves $\alpha_i(t) = min\{M_i + p_i t, b_i + r_i t\}$ and have been assigned guaranteed delays d_i. The system is schedulable since the sum of the arrival curves shifted by the corresponding delay d_i is below the link service $R \cdot t - L$. In order to compute the test for this example, one must check the sum of the arrival curves at all points where the slope changes. For further tests in case a new flow will be checked for admission, all these inflexion points have to be stored. Obviously, the complexity of the tests depends on the chosen traffic specification.*

Thus, packets will always meet their deadlines, even in times of congestion, since packets of aggressive flows that arrive too early at the router are assigned corresponding late deadlines according to the agreed traffic profile.

EDF Fairness: By the first impression one may become convinced that EDF is indeed a fair scheduling algorithm since deadlines are calculated with relation to the expected arrival time of the packet. In the following example, we will look at the resulting service for two backlogged flows.

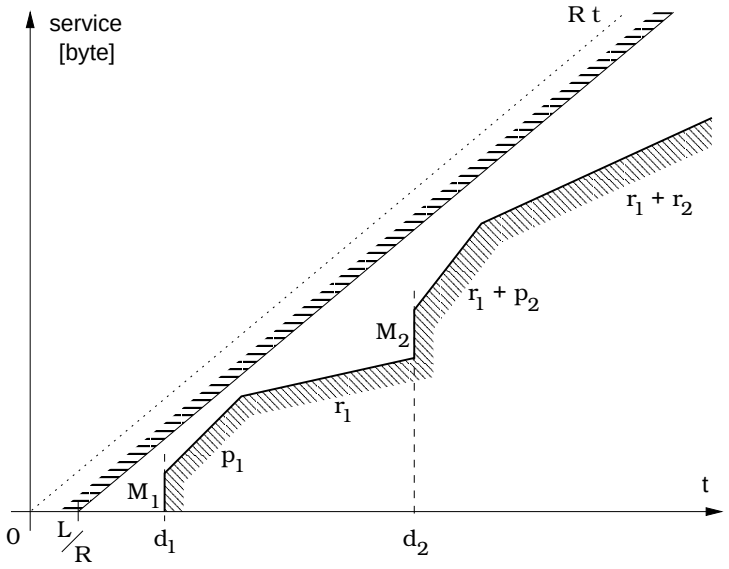

Fig. 6: EDF schedulability test for two TSpec-compliant flows. The system is schedulable since the resulting arrival curve of the incoming traffic is below the link service.

Ex. 4: **(EDF surplus bandwidth distribution)** *Let two flows with (σ, ρ) traffic profiles be continuously backlogged for an arbitrary interval. In Fig. 7, the corresponding schedulability test is shown. Note that a third flow has been admitted which is currently not backlogged. The deadlines for the backlogged flows are calculated according to the (σ, ρ) profiles and the assigned worst-case delay bounds. Since the EDF scheduler is work-conserving, packets are immediately served as soon as the link becomes idle although the expected arrival time of a packet may lie in the future. In this way, the resulting guaranteed services are virtually condensed somewhat in proportion to their reservation because the generated schedule with time stamps for a link under full load has been shrunk to a shorter interval. The resulting actual service is sketched by the dotted line in Fig. 7. Deadlines and the average rates are improved. Although this actually looks very fair, the situation collapses when the third flow becomes backlogged. The other two flows have packets enqueued with deadlines far in the future. The newly backlogged flow however receives very small deadlines for its packets compared with the other flows. As a result, only packets of the newly backlogged flow are served for a long period. Therefore, the short-term unfairness may be extremely high.*

When surplus bandwidth is available, one must adapt the models accordingly that calculate the expected arrival times to bound the short-term unfairness of EDF. Otherwise, flows are handicapped for using surplus bandwidth in the past. Adapting the calculations of deadlines however means to adjust the individual formula for each flow. Contrary to that, this effect is taken into account by a single formula in WFQ – the updates of the virtual time in eq. (2.2) – because the virtual time increases with the normalized marginal rate of all backlogged

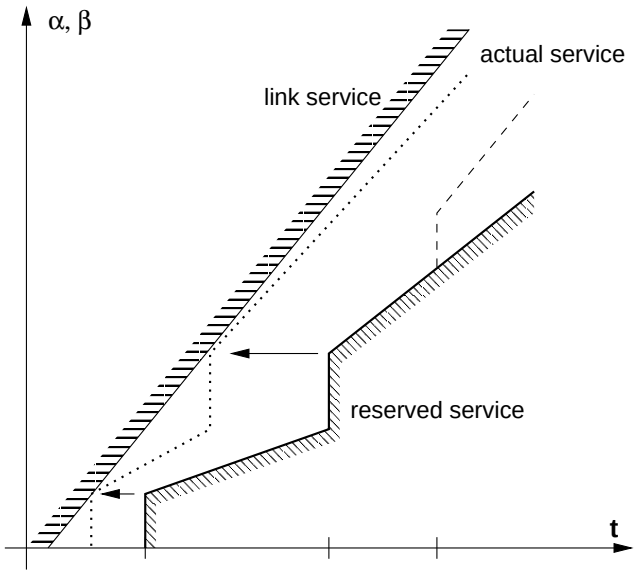

Fig. 7: EDF surplus bandwidth distribution for two continuously backlogged flows.

flows. In conclusion, the advantage of being able to more flexibly trade delay off for bandwidth comes at the price of more complex schedulability tests and per-flow profile adaptations during run-time.

2.2 Preserving QoS

In the preceding section, scheduling algorithms and their schedulability tests have been described almost independently of all other components in the packet processing chain. Moreover, the scheduling tests presented so far only consider whether the scheduler is limited by timing, that is, whether the individual worst-case delays introduced by the server are smaller than the corresponding deadlines of the flows. However, some of the tests, for instance in the FCFS case, actually require the state of the queuing system such as the length of the queue in order to derive a test condition for the scheduler. Besides, every scheduler needs a certain amount of buffer space to guarantee lossless transmission as an option. Consequently, one could in turn think of additional schedulability tests which check whether the schedulability of a system is limited by buffer resources. Finally, for the WFQ scheduler, only delay bounds for the rather unlikely assumption that a packet arrives at an empty queue of an idle flow could be presented in Tab. 1. Obviously, the functionality of the scheduler and the buffer space are closely related to each other and a framework is required to reveal their connection in general to properly make use of queuing and packet scheduling in a real system.

In order to distinguish QoS at least a policer, some queuing space, and a link scheduler (server) are needed as sketched in Fig 8. These components have

to rely on proper QoS classification by a preceding classifier stage to establish per-flow service distinction. How the required buffer space for lossless trans-

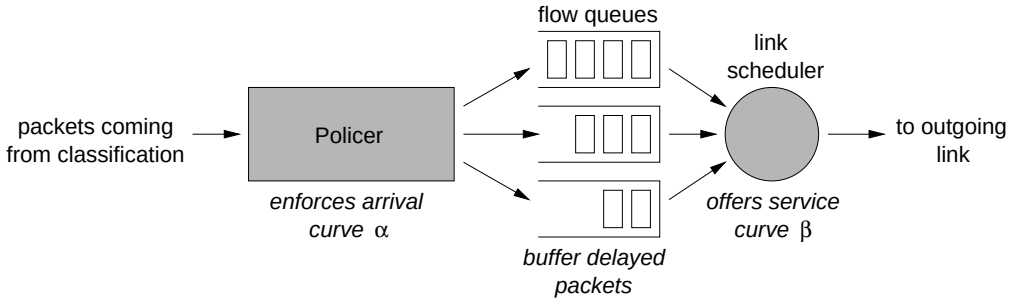

Fig. 8: Infrastructure for enabling QoS distinction and preservation.

mission depends on the guaranteed service and how delays can in turn be determined from buffer allocations, guaranteed service, and traffic specifications will be described in this section. Traffic specifications will be given by arrival curves which have already been introduced informally in the preceding section to describe a traffic envelope. The policer then checks whether incoming traffic is compliant to the corresponding arrival curve α. Service offered by the link scheduler will be specified by a service curve β which describes the least amount of service guaranteed in the worst-case. Arrival curve α and service curve β together can be used to derive bounds for the queuing space, that must at least be provided to guarantee lossless transmission, and bounds for the delay that packets will at most experience in a lossless system.

The following analysis of buffer and delay bounds is based on the framework of service curves which has been applied first by Cruz [40] to communication networks. The nomenclature is adapted and some properties of service curves are shown that have been presented by Le Boudec et al. [103].

Def. 10: **(Arrival function $\mathbf{x}$)** *The arrival function $x(t)$ of a data flow is equal to the number of bits seen on the flow in time interval $[0, t]$ at a defined place in the network.*

Def. 11: **(Input/output functions $\mathbf{x}, \mathbf{y}$ of a system)** *The input function is the arrival function $x(t)$ of a flow or an aggregate of flows seen at an input of a system in the network, e.g., at the input of a router. Accordingly, the output function $y(t)$ is defined to be the arrival function of a flow or an aggregate of flows at an output of a system in the network.*

Def. 12: **(Arrival curve α)** *Given a non-decreasing function $\alpha(t)$ defined for $t \geq 0$, a flow x is constrained by the arrival curve α if and only if for all $s \leq t$:*

$$x(t) - x(s) \leq \alpha(t - s)$$

That means, during any time window of width τ, the amount of traffic for the flow is limited by $\alpha(\tau)$.

That is, since one is not aware of the exact arrival pattern of the packets of a flow, the traffic is rather bounded by a worst-case envelope.

Def. 13: **(Service curve β)** *Given a system S and a flow through S with input and output functions $x(t)$ and $y(t)$, S offers a service curve β to the flow if and only if for all $t \geq 0$ there is some $t_0 \leq t$ such that*

$$y(t) - x(t_0) \geq \beta(t - t_0)$$

That means in particular, a flow backlogged during any interval τ receives at least a flow-through of $\beta(\tau)$.

Worst-case bounds for the flow's backlog at system S as well as the delay experienced by a bit arriving at the system can now be determined.

Theor. 2: **(Bounded backlog b)** *A flow constrained by the arrival curve α passes a system that offers a service curve β to the flow. The backlog $b(t)$ satisfies for all t:*

$$b(t) \leq \sup_{\tau \geq 0}(\alpha(\tau) - \beta(\tau)) \tag{2.9}$$

That is, in order to guarantee lossless transmission of a flow bounded by α and with assured service β, buffer space of at least $\sup_{\tau \geq 0}(\alpha(\tau) - \beta(\tau))$ must be available.

Theor. 3: **(Bounded delay d)** *A flow constrained by the arrival curve α passes a system that offers a service curve β to the flow. The delay $d(t)$ experienced by a bit in the system satisfies for all t:*

$$d(t) \leq \sup_{s \geq 0}(\inf\{\tau : \tau \geq 0 \textit{ and } \alpha(s) \leq \beta(s + \tau)\}) \tag{2.10}$$

Ex. 5: **(Bounds for TSpec flow at WFQ scheduler)** *The backlog and delay bounds have a simple graphical interpretation. Given a flow bounded by the arrival curve α and a system that offers a service curve β to the flow, the maximum horizontal distance between the arrival curve α and the service curve β determines the upper bound for the delay whereas the maximum vertical distance between the curves shows the upper bound for the backlog. In Fig. 9, bounds for a flow n constrained by a TSpec $\alpha(t) = min\{M + pt, b + rt\}$ passing a WFQ scheduler with guaranteed rate r_n, link speed R, and maximum packet length L are derived. The guaranteed service offered by a WFQ scheduler to a flow n is always described by a straight line service curve beginning at $t = \frac{L}{R}$. This offset considers the worst-case delay penalty due to non-preemptive packet scheduling. The slope of the straight line is determined by the reserved rate r_n for flow n.*

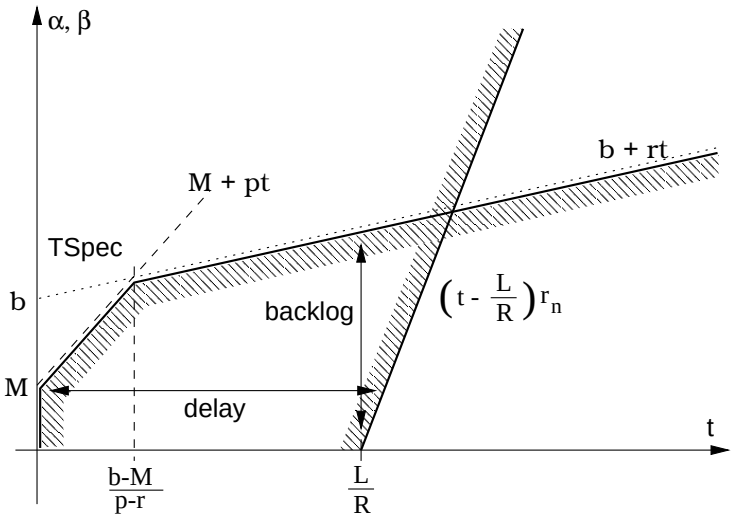

Fig. 9: Bounds on delay and backlog for a TSpec constrained flow with arrival curve $\alpha(t) = min\{M + pt, b + rt\}$ and a WFQ scheduler with guaranteed rate r_n, maximum packet length L, and link rate R.

Delay $d(t)$ and backlog $b(t)$ for flow n are bounded by:

$$d(t) \leq \begin{cases} \frac{L}{R} + \frac{M}{r_n} & , p \leq r_n \\ \frac{L}{R} + \frac{M}{r_n} + \frac{b-M}{p-r}(1 - \frac{p}{r_n}) & , p > r_n \end{cases}$$

$$b(t) \leq \begin{cases} b + \frac{L}{R} \cdot r & , \frac{L}{R} \geq \frac{b-M}{p-r} \\ M + \frac{b-M}{p-r}(p - r_n) + \frac{L}{R} \cdot r_n & , \frac{L}{R} < \frac{b-M}{p-r} \text{ and } p > r_n \\ M + \frac{L}{R} \cdot p & , \text{otherwise} \end{cases}$$

Obviously, the special case for the worst-case delay $\frac{L}{R} + \frac{L_n}{r_n}$ experienced by a packet of arbitrary length L_n arriving at an empty queue of an idle flow given in Tab. 1 is included in the results of this example since $L_n \leq M$ is true per definition.

Ex. 6: **(Bounds for TSpec flow at EDF scheduler)** *A flow bounded by a TSpec $\alpha(t) = min\{M + pt, b + rt\}$ is assigned a deadline d_n at an EDF link scheduler with link rate R and maximum packet length L. Since the EDF scheduler assures that every incoming packet will finish service at most d_n seconds after its arrival at the system, the assigned service curve grows impulse-like from zero to infinity at time d_n, see Fig. 10. Note that the starting point of the impulse is variable, since the deadline can arbitrarily be chosen as long as the system remains schedulable according to eq. (2.8) and $\frac{L}{R} < d_n$. Again, the term $\frac{L}{R}$ is caused by the non-preemptive packet service. Opposed to that, the starting point of a service curve for a WFQ scheduler is always fixed at $t = \frac{L}{R}$ whereas the slope of the service curve – the reserved rate – may arbitrarily be chosen.*

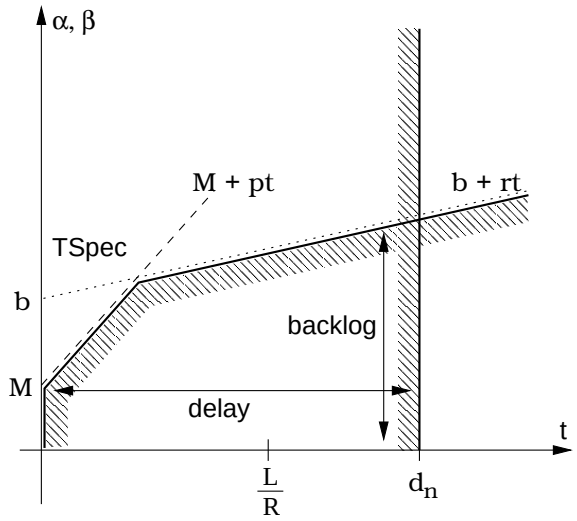

Fig. 10: Bounds on delay and backlog for a TSpec constrained flow with arrival curve $\alpha(t) = min\{M + pt, b + rt\}$ and an EDF scheduler with guaranteed deadline d_n, maximum packet length L, and link rate R.

Obviously, the delay bound for the EDF system must be the deadline and the backlog bound for flow n becomes

$$b(t) \leq b + r \cdot d_n$$

Since a flow passing a system experiences a variable delay, the burstiness of the resulting flow potentially increases. The arrival curve of the flow leaving the server can be described by the following theorem.

Theor. 4: (Output flow constraint) *A flow constrained by the arrival curve α passes a system that offers a service curve β to the flow. The outgoing flow is constrained by the arrival curve α^**

$$\alpha^*(t) = \sup_{\tau \geq 0}(\alpha(t + \tau) - \beta(\tau)) \tag{2.11}$$

In summary, the following can be done with service curves as introduced in this section:

- *Given a set of flows constrained by α_n and service reservations β_n:* Derive delay bounds with eq. (2.10) and buffer requirements with eq. (2.9). Check whether deadlines will be met if deadlines are defined. Check whether sufficient memory is available.

- *Given a set of flows constrained by α_n and deadlines:* Use eq. (2.10) to derive suitable service reservations in order to keep the deadlines for each flow. Determine the buffer requirements with eq. (2.9) and check whether there is sufficient memory available.

That is, beside the usual schedulability condition that tests whether a system is bounded by timing, there is now the additional option to define a condition that tests whether the system is bounded by buffer space. In other words, the accurate mode of operation for preserving QoS does not only depend on the scheduler which guarantees service curves β_n and the policer that enforces arrival curves α_n but also on the queue manager that must have sufficient memory available for individual flows.

In the end, one further theorem enables the analysis of whole networks of nodes. The following theorem is required to explain some assumptions of available services which are described in the next section.

Theor. 5: **(Concatenation of nodes)** *A flow passes two systems S_1 and S_2 in sequence which offer service curves β_1 and β_2 respectively. The resulting service offered by the concatenation of the two systems can be described by a service curve β'*

$$\beta'(t) = \inf_{s:0\leq s\leq t}(\beta_1(t-s) + \beta_2(s))$$

Ex. 7: **(Concatenation of service curves)** *Theorem 5 has a simple graphical interpretation for piecewise-linear, convex curves. In order to derive the resulting service curve for the concatenation of two service curves with this property, the linear segments of both curves must be put together end to end sorted by increasing slope. In the special case of the concatenation of two WFQ-like service curves we end up with adding the delay offsets and using the minimum rate as shown in Fig. 11.*

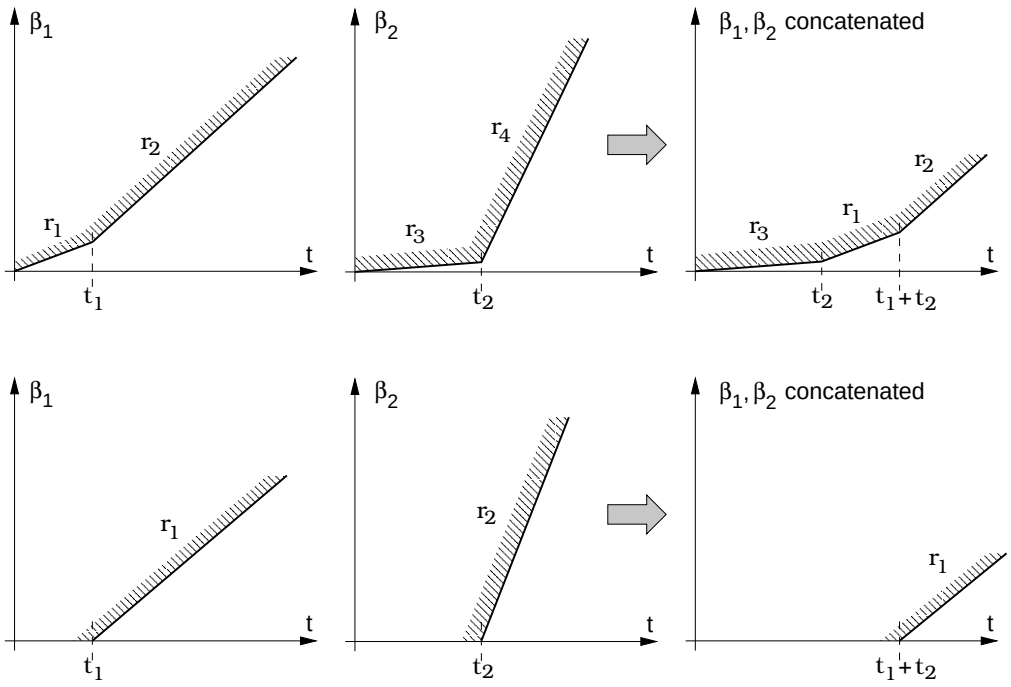

Fig. 11: Some examples for the concatenation of service curves .

2.3 Available services

In this section, the most common end-to-end service classes provided by today's Internet are reviewed. *Integrated Services* are well suited for reliable real-time communication and provide a connection oriented-like distinction between flows. *Differentiated Services* define a relative priority scheme that distinguishes a fixed number of service classes which represent aggregates of flows. If the network does not provide any differentiation of traffic, flows will be forwarded by *best-effort*. Open issues under research in order to implement services efficiently are outlined at the end of this section.

2.3.1 Best-effort service

This type of service does not guarantee or define any bounds, reliable service, or QoS at all. All packets are handled in the order they arrive at the system as long as there are sufficient resources available. The system does its best to forward all incoming traffic. Flows are not distinguished and therefore not protected against each other. Service is never denied but potentially degrades with higher load for all participants. Despite these limitations, best-effort service is still a suitable solution and can be found in most of today's Internet routers since it can simply be provided. All incoming packets are stored in a FIFO-organized queue and served in FCFS order. No admission or schedulability tests are performed. Congestion may be avoided and/or resolved by the queue manager, e.g. with a RED policy, or by overprovisioning of network resources. The latter solution is especially feasible in a distribution network of a service provider.

2.3.2 Integrated Services (IntServ)

IntServ [20] is characterized by resource reservation. That is, flows which may represent, for instance, customers, applications, or whole organizations must set up paths through the network and reserve resources at each networking node. That is, routers are expected to maintain per-flow state information. The resource reservation protocol RSVP [22] is usually applied as a signalling protocol for this purpose. Traffic is policed at the edge of the IntServ network and may be reshaped to a defined profile within the network. The underlying service scheme works as follows. Each network node i in the path is expected to offer a service which resembles the service of an ideal fluid server. In practice, a node's service is specified by a rate and two error terms C_i and D_i which define the deviation from the service of an ideal fluid server as sketched in Fig. 12. The term C includes all rate-dependent errors whereas D considers additional constant offsets. C is at least the maximum length of a packet since a packet transmission cannot be preempted. Moreover, each node along the path uses the same service rate R. Thus, the impact of the error terms is additive along the path, see Theorem 5 and Ex. 7, and the destination router sees a resulting service curve

$$\beta'(t) = \max\{0, (t - \frac{\sum C_i}{R} - \sum D_i) \cdot R\}$$

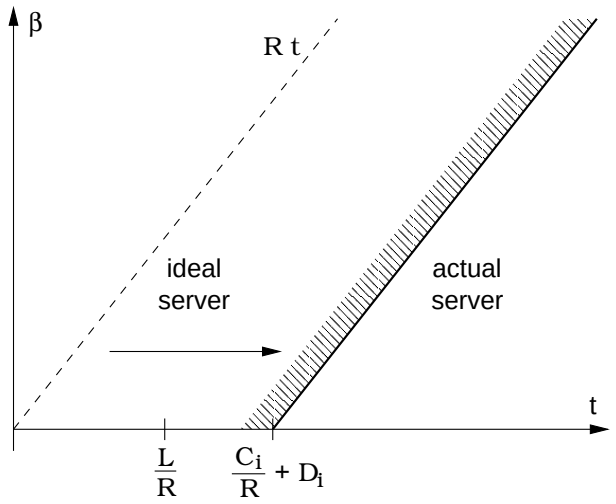

Fig. 12: Service offered by a network node i in an IntServ path supporting a service rate R. The error term C_i represents rate-dependent delay effects whereas D_i stands for fixed delay offsets. C_i must be greater or equal the maximum packet length L.

A path is set up by sending a *PATH* message from the source to the destination router together with a flow descriptor and a traffic characterization specified by a TSpec (Fig. 13). The route of the path has been determined beforehand by a routing protocol and is not part of any IntServ/RSVP procedure. As the path message advances along the route to the destination, each intermediate node adds its specific delay characteristics C_i and D_i to the message. Together with the TSpec of the source traffic, the destination router can now determine the service rate R which is needed to achieve a suitable end-to-end delay due to the delay characteristics of the path in a similar way as in Ex. 5. The requested rate R is advertised by a *RESV* message that is directed from the destination to the source and which follows the same route as the PATH message in reverse direction. Each intermediate node can now calculate the buffer space required to support the desired service since the local service curve is known by the parameters C_i, D_i, and R. Moreover, a traffic characterization can be derived from the TSpec at the source and the service curves of the nodes between source and the current node. The current node is able to reconstruct the delay properties of these nodes since their cumulative error terms have been broadcast with the preceding PATH message. By applying a schedulability test a node may detect that it is not able to support the rate R or that not enough buffer space is available. Appropriate teardown messages reset the nodes in the path in this case.

Two types of service are distinguished:

- ***Guaranteed service [140]:*** This service guarantees the reliable delivery of conforming traffic with fixed delay bounds as it would be expected from a dedicated constant bit rate line. Excess traffic should be forwarded as best-effort traffic.

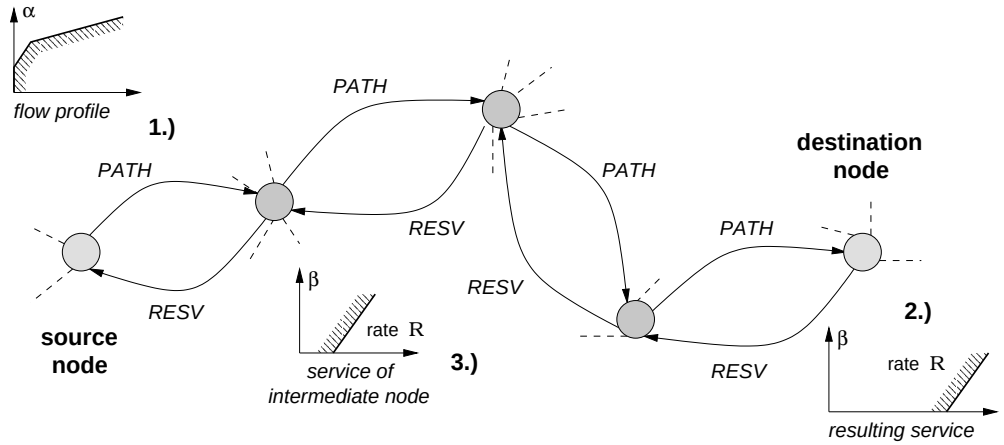

Fig. 13: RSVP signalling for path establishment and resource reservation. 1.) A *PATH* message collects the delay error terms of the intermediate network nodes from the source to the destination node and carries the flow specification of the source. 2.) The destination allocates a service rate R which is broadcast by a *RESV* message. 3.) Each node uses the flow profile, the service rate R, and the announced delay properties of the path to reserve buffer space.

- ***Controlled-load service [166]:*** Controlled-load service offers the delivery of traffic by an enhanced best-effort service. It approximates the service an unloaded best-effort network would provide, i.e. without the appearance of congestion or heavy load. However, there are no quantitative delay or loss guarantees. This type of service thus allows the provider to more flexibly allocate resources along the path. Note that the signalling procedure described above is still required for the controlled-load service, i.e., the traffic is in particular expected to conform to the TSpec. However, non-conforming traffic is not considered to be unusual and the router will be obliged to forward excess traffic on a best-effort basis if sufficient resources are available.

The advantages of IntServ/RSVP are:

- Perfect isolation of flows by using guaranteed service classes.
- Hard guarantees of delays and reliable transmission are especially suited for real-time communication.
- The “one size fits all” approach for resource reservations along a path simplifies the resource allocation which can be performed in a single run along the path.

The disadvantages of IntServ/RSVP are:

- There is a signalling overhead – even for the controlled-load service – which may noticeably delay short-time flows and require considerable computing.
- All routers must maintain per-flow state information which may be inadequate for backbone routers.

- The infrastructure of all networking nodes that take part of IntServ must support decent scheduling, signalling, and classification of packets.

- On the one hand, the guaranteed service type has no concept of sharing resources. On the other hand, the controlled-load service allows sharing of resources but without supporting any quantitative guarantees.

- The "one size fits all" approach for resource reservations along a path prevents individual resource allocations, for instance, dependent on the load of a router so as to flexibly compensate individual delays.

- IntServ will not explicitly prevent packet reordering if excess traffic appears for the corresponding flow.

- There are no guarantees on the end-to-end jitter.

2.3.3 Differentiated Services (DiffServ)

Service levels in DiffServ [17] are based on relative priorities with different sensitivities to delay and loss, but without quantitative guarantees. DiffServ does not require signalling to take place for each flow. There may be a static SLA with the provider that specifies the number of different service classes and the amount of traffic allowed in each class. Dynamic SLAs may be negotiated by using an enhanced version of RSVP. Opposed to IntServ/RSVP, the resource reservation is then initiated from the source and not from the destination node. Note that the provider may arbitrarily assign network resources to the different classes. At the ingress nodes of the DiffServ network packets are classified and marked in the DS-field – six Bits of the redefined ToS field of the IP header – with the corresponding class identifier, the so-called codepoint. Alternatively, the originating traffic host is allowed to set the DS-field in order to encourage the corresponding service. It is the customer's responsibility to share bandwidth between flows within the same class. Moreover, if too many flows share the same class, the service may degrade to best-effort. Within the network, the DS-field is used at each node to determine the corresponding service class, the so-called Per-Hop forwarding Behavior (PHB). Only two standardized PHB should be supported by a DiffServ-capable router. However, the provider may define a limited number of own service classes within a DiffServ domain. Thus, a PHB usually processes an aggregate of flows and not individual flows. It is tolerable to some extent that a DiffServ aggregate passes a DiffServ-incapable node. The DS-field is simply ignored by this node and best-effort service is applied.

Note that being DiffServ-compliant only means that a router has resources to support packet classification, policing, and marking based on code-points and perhaps other header fields if the router is placed at the edge of the DiffServ network. It does not imply any quantitative service guarantee or supported traffic profile. The number of QoS classes, the traffic profiles, and the kind of service to support are the subjects of individual service level agreements between cus-

tomers and the provider of the DiffServ domain. The format of these agreements is not fixed by any standard.

Per-Hop-Behaviors that should be supported by a DiffServ-capable router are:

- ***Expedited Forwarding (EF, or Premium Service, [89]):*** This single class provides the relatively highest level of service in terms of delay and loss. Excess traffic will be dropped at the ingress nodes of the DiffServ domain. EF traffic has higher priority than any other traffic from other PHBs. The provider must limit the amount of bandwidth admitted to EF if a static priority scheduler is employed so as to not completely starve other classes. EF must have a well-defined minimum departure rate independent of the state of the node.

- ***Assured Forwarding (AF [80]):*** AF defines up to four classes with up to three drop precedence subclasses within each class which should provide better reliability than best-effort service. Thus, AF consists of up to 12 different codepoints. More AF classes and drop precedence levels may be defined for local use within an administrative domain. Non-conforming traffic is demoted to a higher loss precedence and not necessarily dropped. Packets with different drop precedence levels maintain their relative order within a class. The parameters of the dropping algorithm must be independently configurable for each packet drop precedence and for each AF class. There are no regulations concerning the relative priority between the AF classes. A minimum amount of forwarding resources must be reserved for each implemented AF class. There are no rules how to handle surplus resources.

The advantages of the DiffServ approach are:

- Multi-field classification must only be performed at the ingress nodes of a DiffServ domain. Due to the DS-field, classification reduces to a table lookup within the domain. This resembles the network structure since ingress routers usually work at lower bandwidths than core routers and are therefore able to spend more time with classification and policing.

- The complexity of the scheme scales with the small number of possible classes and not with the number of flows.

- Soft bounds which are negotiated between customers and service providers allow the providers to exploit their equipment by sharing resources and individual reservations at each router.

- A graceful service degradation will be performed if an AF class exceeds its profile.

- There is no reordering of packets.

- The EF class is served in isolation from the AF classes, and AF classes are treated in isolation from each other.

The disadvantages of DiffServ are:

- There are no guarantees on delay and the level of loss.
- A flow is not protected against a greedy flow within the same service class.
- Since resource reservations for the PHBs are not explicitly communicated and balanced between the routers of a domain, the relative priority between the classes may fluctuate from hop to hop along a path.
- An application may not be aware whether a particular service is being delivered because there are no explicit feedback mechanisms which, for instance, adapt the policer in dependence on the system's state. This is why resource availability signalling has been proposed recently [86].

2.3.4 Open issues in providing Quality of Service

Since IntServ and DiffServ only provide a framework for the preservation of QoS, implementations suffer from a variety of still unsolved problems and rarely achieve a high utilization of resources. Some of the most often discussed points are:

- ***Traffic engineering:*** Some congestion situations will be avoided if traffic is better distributed over the network resources. Since most of the routers today only consider the shortest path to a destination, it is often the case that some few routes are congested while others are lightly loaded. A more elaborated routing scheme would find a route which fulfills the QoS requirements of a given request while balancing the utilization of the network.
- ***Service allocation:*** This issue points to the problem of pricing the Internet. For instance, look at a customer's domain connected to a DiffServ domain. Since there is only a small number of service classes and it is the customer's responsibility to rationally balance the usage of the classes according to an SLA, individual hosts should not access the DiffServ domain directly. An arbiter instance, such as a bandwidth broker, should regulate the access to the different DiffServ codepoints and configure markers and policers accordingly.
- ***IntServ over DiffServ:*** Smaller LANs such as enterprise networks are able to afford the sophisticated QoS distinction of IntServ. IntServ however is not a feasible solution for a WAN interconnecting IntServ capable enterprise networks. IntServ flows should thus be aggregated into DiffServ classes in a way that the service appears as a virtual leased line to the IntServ flows. This topic has recently been discussed in [15].
- ***TCP-aware services:*** Current Internet technology for providing QoS can be considered to be TCP-hostile [86]. For instance, the TSpec used to specify a flow in the IntServ framework that corresponds to a conjunction of token buckets may abruptly change from marking packets as conforming to dropping packets for several packets in a row. This may result in an excessive degradation of

the average usable bandwidth due to enlarged retransmission timeouts. Thus, a marker which randomly marks packets with some probability is suggested in [48]. A similar effect appears with queue managers that use a rather deterministic approach to drop packets such as a tail-drop policy. In this case, it is very likely that packet from several TCP flows are dropped. These flows reduce their transmission rate at the same time. These consequences can again be avoided by some level of randomization in choosing packets to drop, e.g. by applying RED [54]. Last but not least, the behavior of a TCP flow additionally depends on the availability of the corresponding feedback mechanism and vice versa, see [99]. That is, the service quality of the forward direction influences the level of service needed for the backward direction. How these results affect the stability of the whole network and which amount of resource reservation would prevent aberrant behavior is not yet understood.

- ***Individual resource allocations and end-to-end behavior:*** The IntServ/RSVP service model uses a constant rate in order to reserve resources along a path in the network. The network however could be better utilized by individually reserving resources dependent on the router's load so as to compensate long delays at one node by short delays at other nodes. The same problem arises for the provider of a DiffServ domain that has to route traffic from different customers. The effects of a given reservation on the end-to-end delay and buffer requirements are however well understood for a variety of traffic characteristics and heterogeneous networks using diverse packet schedulers. Networks of schedulers that reshape traffic at each node to a given profile are analyzed in [59, 171]. Heterogeneous networks that do not necessarily depend on reshaping are investigated in [151]. In addition, time-varying rate allocations are considered in [63]. Reshaping at each networking node has the advantage that the traffic conforms to the profile at every node. This way, buffer requirements are balanced along the path. The average delay may be affected by reshaping but not the worst-case delay as long as only packets are delayed which are ahead of their profiles. If no reshaping is applied, the buffer requirements will potentially increase toward the destination node and the jitter may be in the same order of magnitude as the end-to-end delay because the burstiness of the flow raises from hop to hop. In any case, the end-to-end delay analysis derives tighter delay bounds than simply adding the delay bounds of all nodes in the path.

3

IP packet processing: Algorithm-architecture trade-offs

In this chapter, a new application area for access networks is introduced that we call *multi-provider/multi-service* access networks. For this kind of network this chapter is devoted to an evaluation of packet processing algorithms and architecture blocks that will be used by dedicated processing elements at the network access points. In order to achieve comprehensive results, the following methodic approaches are applied:

- A new service scheme is defined which is derived from the requirements of multi-service access networks. It combines the advantages of existing services by providing quantitative guarantees for flows as well as qualitative guarantees for aggregates of flows.

- For the first time a combined evaluation of processing stages which are responsible for the Quality of Service (QoS) behavior of network access points, namely policing, queuing, and link scheduling, is presented.

- A design space exploration of suitable algorithms and hardware resources is performed by co-simulation of performance models for algorithms and timing models of hardware building blocks. Thus, the usage of networking resources can be appraised more flexibly and exhaustively than it has been done up to now with static complexity analysis or existing hardware platforms.

  We are thus not only able to show how the combination of policing, queuing, and link scheduling affects the behavior of an access network element in terms of Quality of Service (QoS) but we also reveal the expense in terms of hardware resources required to implement sophisticated QoS preservation.

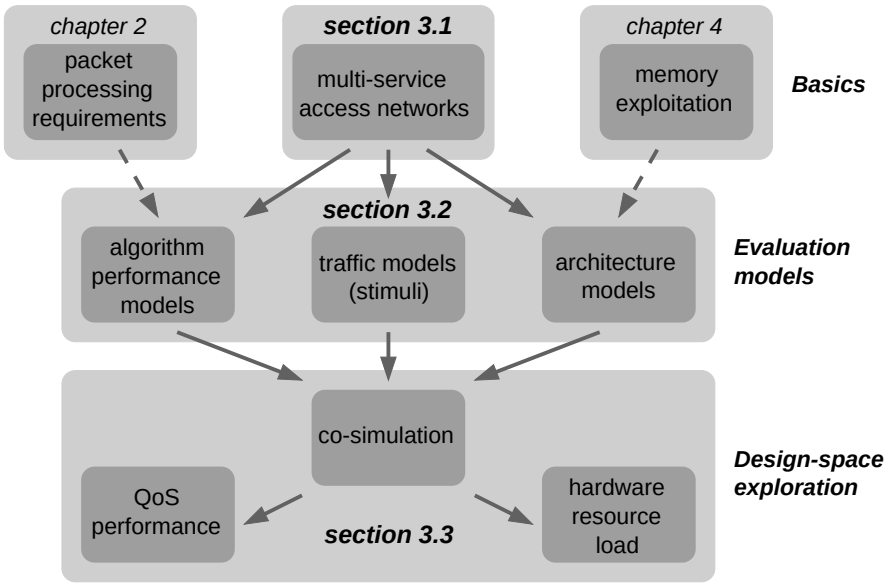

Fig. 14: Chapter overview.

This chapter is structured as displayed in Fig. 14. In the next section, multi-service access networks are defined and their requirements are determined. The effects on our suggested QoS scheme are derived and a suitable access node architecture is shown. Based on the definition of multi-service access networks and the results of the preceding and the next chapter, the settings for the evaluation – algorithms, hardware blocks, and traffic models – are discussed and determined in Section 3.2. Algorithms are modeled without any architectural assumptions or constraints whereas the hardware architecture models reproduce timing and load of hardware resources. The two types of models cooperate by exchanging statistics about their inner state during run-time. Moreover, the characteristics of the network traffic which is used to stimulate the access network node are described. The results of our design space exploration by simulation of different combinations of algorithms with diverse input traffic patterns and various architecture blocks is presented in Section 3.3. Finally, an overview of related work can be found in Section 3.4 at the end of this chapter.

3.1 Background

The concept of multi-provider/multi-service networks is introduced in this section. The consequences on the provision of QoS and on a suitable architecture for a dedicated packet processor are derived. A new service scheme is defined that combines the advantages of IntServ and DiffServ concerning the requirements of multi-service access networks.

3.1.1 Multi-provider/multi-service access networks

In the Internet one can see the evolution of various content providers, for instance, for news, voice telephony, and multi-media streams such as movies. Currently, the access link from the customer to the network is only used for accessing a single kind of contents at a time, for instance, the customer either uses the link for telephony or for accessing the Internet at a time. Moreover, the customer's traffic is delivered independently of the contents by the service of the underlying Internet technology, i.e., with best-effort. As soon as access link technologies are available for a broad range of customers that provide an access to the network with several MBits per second such as cable modems and DSL lines, the distinction of QoS will become more and more important because access links will be used for concurrent connections to several content providers with different tolerable QoS levels. An enterprise, for instance, may have several Service Level Agreements (SLAs) for telephony, for accessing the Internet, and for reliable interconnection of Virtual Private Networks (VPNs) of the enterprise that are distributed all over the globe. A private customer may access movies, Internet contents, and telephony services from different providers. The resulting network structure is sketched in Fig. 15. The customer will have

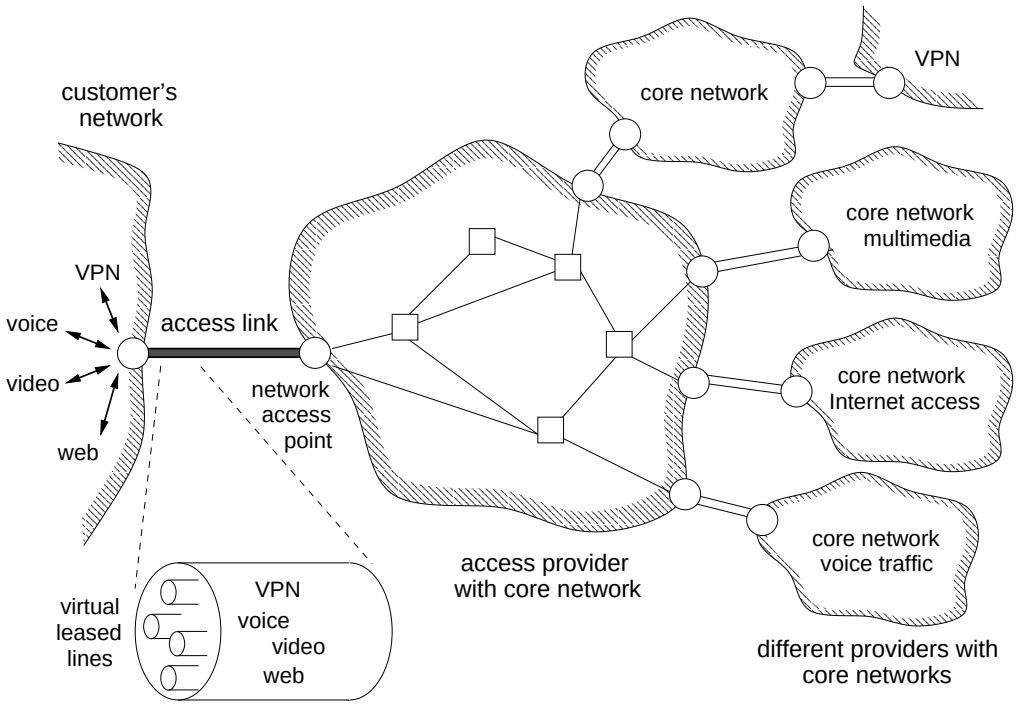

Fig. 15: Network structure with a single access link and multiple content providers.

an SLA with the provider of the access link which guarantees the somewhat reliable distribution of traffic to different content providers. The customer may have additional SLAs with these providers. Alternatively, the access link and different contents/services may be supplied by the same provider. In any case, the customer will use several virtual leased line-like traffic classes at the net-

work access point. Traffic classes may be further divided into subclasses that consider the type of application, the origin, etc. This shift of the objective from the provision of raw bandwidth to the delivery of services at access networks is also promoted in [113].

3.1.2 Provision of QoS in multi-service access networks

The assumption of a multi-service access network has several effects on issues concerning the preservation of QoS. Two types of services – green and yellow service – are therefore defined to consider the special requirements of this kind of network. The new service scheme combines the advantages of the existing IntServ and DiffServ approaches. Based on the characteristics of packet processing requirements discussed in the preceding chapter, traffic profiles, guaranteed bounds, and reservations are adapted to suit the new service scheme.

Our network processor is aimed to bundle and arbitrate traffic streams at the bound of the customer's network to the bottle-neck link – i.e., the access link. It is assumed that the customer's traffic shows a high amount of non-reactive flows in the sense that these flows do not respond on packet drops by adaptively reducing their bandwidth. Examples for such flows are voice and video traffic streams without feedback control. As a consequence, we are able to use policing elements which are known to be TCP-hostile. The policer thus carries the main responsibility to limit flows to a given profile in our solution.

- ***Service classes:*** Since the customer may have several SLAs with different content providers, a minimum service for distinct flows to each provider should be guaranteed during times of congestion on the access link. A *green service* QoS class will be introduced later to take care of these reservations. Moreover, there should be maximum bounds for the overall traffic to a provider during light load of the access link. These maximum bounds are constraints of so-called *meta-classes* because they only concern the type of service for the virtual leased line to the corresponding provider. The service quality of a meta-class is defined by a so-called *yellow service* QoS class. Within a meta-class there may be several green service classes that may correspond to traffic from different application types, various users, diverse origins, etc. These requirements of a multi-service access network motivate the following new combination of QoS classes.

Def. 14: **(Green service)** *Green service is provided to classes that are specified by minimum traffic profiles within a meta-class. The green service includes lossless transmission of data with guaranteed delay bounds as long as a flow complies with its traffic profile. No packet reordering is allowed.*

Green service offers the same quality of service as IntServ's guaranteed service [140]. Contrary to the IETF specification, no particular traffic profiles or service curves are assumed.

Def. 15: **(Yellow service)** *Yellow service is provided to a flow that exceeds its traffic profile for green service but complies with the traffic profile of its meta-class.*

Quantitative guarantees need not be given. Depending on the resource allocation the service may vary from best-effort to green service. In particular, a flow may experience loss. The packet order within a flow however must be preserved, not only among yellow packets but also among yellow and green packets.

Green service corresponds to an IntServ-like service whereas yellow service relatively weighs the service for meta-classes in a DiffServ-like style. Graceful service degradation is enabled by adaptively changing from green to yellow service and packet reordering is forbidden in a similar manner as it is defined by DiffServ. Opposed to DiffServ quantitative guarantees are given for green service including lossless transmission, but yellow service need not be defined separately for each green service class. In this way the advantages of both IntServ and DiffServ are combined in the suggested new service scheme. On the one hand, flow isolation and lossless transmission is offered as IntServ does and on the other hand exploitation of resources by sharing is facilitated as DiffServ permits. In this way, the system complexity may be chosen to scale with a small number of meta-classes or with a higher number of flows within meta-classes. Note that surplus bandwidth need not be shared in a fair manner. The resulting new hierarchic service allocation can be interpreted as sketched in Fig. 16.

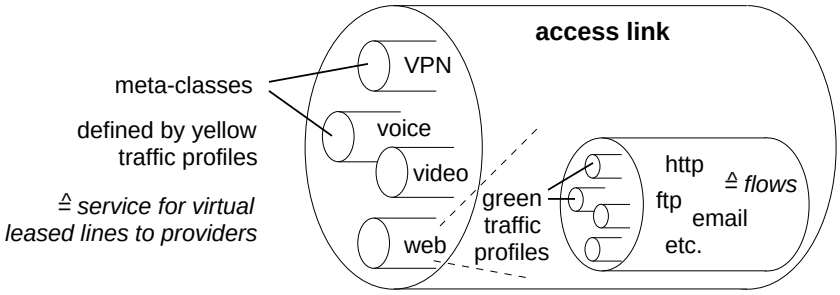

Fig. 16: Service allocation in multi-provider access networks.

- ***Traffic specifications:*** For our evaluation, traffic profiles for green and yellow services are specified by (σ, ρ) models with burstiness σ and rate ρ (see subsection 2.1.3) as an example. Index g denotes a green service profile and index y a yellow service profile respectively. The following constraints must be kept for all meta-classes m since each meta-class profile should at least accommodate its included green traffic profiles. N_m denotes the number of green service classes within meta-class m:

$$\sum_{i=1}^{N_m} \sigma_{g,i} \leq \sigma_{y,m}, \qquad \sum_{i=1}^{N_m} \rho_{g,i} \leq \rho_{y,m}$$

Nested token buckets are used (see subsection 2.1.3) to meter a meta-class and its included classes. A packet will be marked as conforming to a green service profile if a sufficient number of tokens are available in the flow's bucket and in

the bucket of the meta-class. If there are only enough tokens in the bucket of the meta-class but not in the flow's bucket, the packet will be marked as conforming to the meta-class profile (yellow). Otherwise, the packet will be dropped.

Simple traffic profiles are chosen by using the (σ, ρ) model to limit the number of token buckets that must be supported. However, one can easily extend our nested token bucket model in order to meter TSpec-compliant flows.

- ***Choice of the link scheduler:*** Since the QoS and the delay bounds for a flow should not depend on the admission and behavior of other flows, a dynamic priority-driven scheduler is the first choice. Comparing an Earliest Deadline First (EDF)-based system with a Weighted Fair Queueing (WFQ)-based system there are two possible solutions. As described in the preceding chapter, EDF tends towards punishing flows that made use of surplus bandwidth in the past. In order to prevent this behavior, one could implement a policing element that only forwards traffic to flow queues that complies with green profiles. The policer could separate yellow traffic from green traffic. Yellow traffic would be served from queues which are distinct from green queues and which are characterized by own service profiles. Opposed to that, a WFQ system will be able to enqueue green- and yellow-marked packets in the same queue if they belong to the same flow. No additional service profile must be maintained for yellow traffic because surplus bandwidth is inherently shared fairly in proportion of the reservation for green traffic. The two choices are shown in Fig. 17. Although the EDF

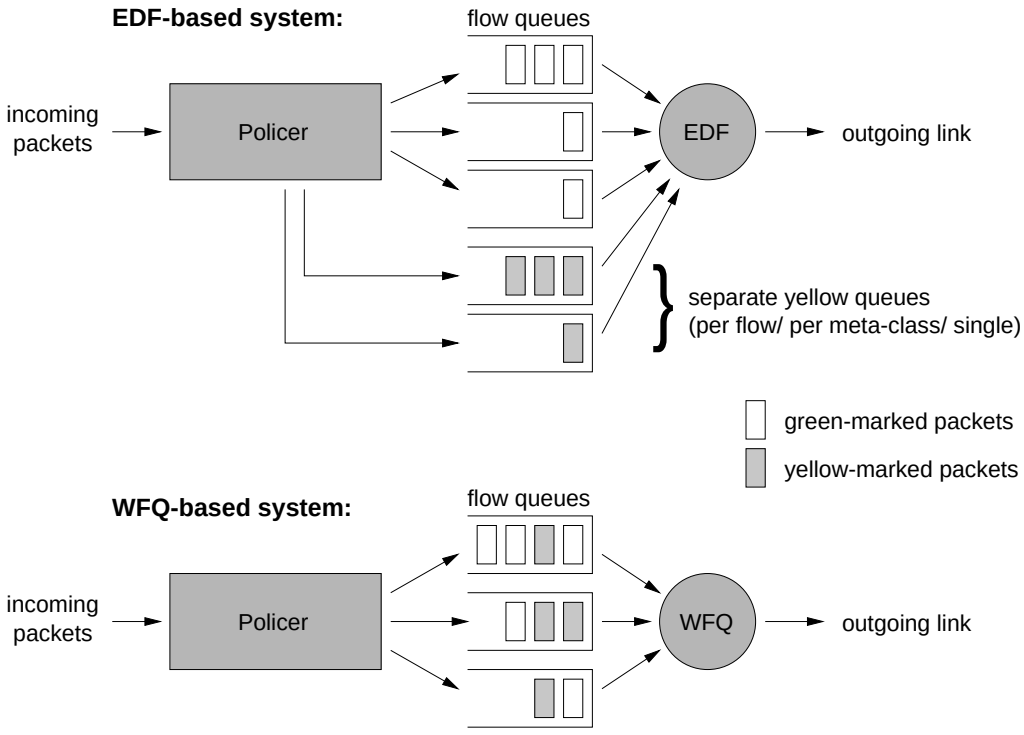

Fig. 17: Packet scheduling for multi-provider/multi-service access networks.

system allows a more flexible distribution of surplus bandwidth that need not

necessarily be fair it also does not prevent reordering of packets. An alternative EDF-based system would enqueue packets in the same way the WFQ-based system does. However, the calculation of individual deadlines would have to be adaptively adjusted at the occurrence of surplus bandwidth as described in Section 2.1.5. Therefore, a WFQ-based system is chosen since we rely on the in-order transmission of packets and do not want to cope with adaptively adjusting per-flow deadline calculations during run-time. A positive side-effect of the WFQ-based system is the fair sharing of surplus bandwidth in proportion to the reserved green service. The green service is guaranteed by corresponding weights $\Phi_{g,i}$ at the WFQ scheduler and the yellow service is bounded by the policer and the queue manager. A more sophisticated solution would be to use a hierarchy of WFQ schedulers. The top level would weigh the yellow services of the meta-classes against each other and the underlying level of WFQ schedulers would distribute the green service within meta-classes. However, since delay and fairness properties are accumulated through the levels of the hierarchy ([13]) and the delay penalty of a single level due to non-preemptive packet transmission may already be noticeable with our settings, the scheduler is restricted to a single level.

- ***Guarantees, reservations, and SLAs:*** Since a multi-provider access node with a combination of a policer based on nested token buckets and a single WFQ scheduler is implemented, the following services can be provided. Green traffic profiles are defined by $(\sigma_{g,i}, \rho_{g,i})$ traffic envelopes for all flows $i \in [1, 2, \ldots, N_m]$ and all meta-classes $m \in [1, 2, \ldots, M]$ where N_m is the corresponding number of flows in meta-class m. Yellow traffic profiles for all meta-classes are given by $(\sigma_{y,m}, \rho_{y,m})$ envelopes. WFQ weights for green services and all flows i are specified by $\Phi_{g,i}$. The bounds can be derived by applying the theorems 2 and 3 in the same way as described in example 5.

  Guaranteed delay bound for green service: A green service class with a traffic profile $(\sigma_{g,i}, \rho_{g,i})$ and a WFQ weight $\Phi_{g,i}$ is guaranteed a delay bound $d_{g,i}$ of

$$d_{g,i} = \frac{L}{R} + \frac{\sigma_{g,i}}{\Phi_{g,i} \cdot R} \tag{3.1}$$

  and lossless transmission as long as the green traffic profile is kept where L is the maximum packet length at the link and R is the link rate. WFQ weights in the range $\Phi_{g,i} \in (0, 1]$ are assumed.

  System schedulability: The system is schedulable and keeps the delay bounds for green services if

  - the flows are within their green traffic profiles,
  - a flow's assigned deadline – if defined – is greater or equal to the corresponding delay bound $d_{g,i}$,
  - the system is stable

$$\sum_{m=1}^{M} \sum_{i=1}^{N_m} \Phi_{g,i} \leq 1, \qquad \rho_{g,i} \leq \Phi_{g,i} \cdot R \quad \forall i$$

where M is the number of meta-classes and N_m the number of flows within meta-class m respectively,

– and there is enough buffer space B available:

$$B = \sum_{m=1}^{M} \sum_{i=1}^{N_m} \left(\sigma_{g,i} + \frac{L}{R} \cdot \rho_{g,i} \right) \tag{3.2}$$

Delay bound for yellow service: As soon as a flow i violates its green traffic profile, the worst-case delay bound drops to the bound $d_{y,m,i}$ which is determined by the characteristics of the meta-class m to which flow i belongs:

$$d_{y,m,i} = \frac{L}{R} + \frac{\sigma_{y,m}}{\Phi_{g,i} \cdot R}$$

The packets are still served in-order and packets marked as green do not experience loss. Yellow-marked packets however may suffer loss.

By varying the allocation of buffer space for yellow service one may shift the main task of congestion avoidance from the policer to the queue manager and vice versa. Suppose that as much buffer space is allocated to accommodate the sum of all yellow profiles. In this case, the system would rather be bounded by the profiles enforced by the policer than by limited buffer space. The latter case could however still appear since the sum of the yellow profiles usually represent an overreservation of the access link. Another buffer allocation scheme could reserve as much buffer space for yellow service to fulfill the needs of the maximum yellow profile only. In this way, we would attach high importance to the sharing of buffer resources controlled by the queue manager.

The SLA with the provider of the access link should at least contain the specification of a guaranteed rate – the bandwidth of the access link – and a bound of the burstiness of the traffic aggregate. The SLA could also be very detailed, specifying bandwidth and burstiness bounds for every meta-class and every distinct flow therein. Recall that the burstiness seen by the access provider is not the same as the burstiness of the policed traffic profile. The link scheduling stage may potentially introduce additional burstiness as it has been derived in Theorem 4 in the preceding chapter. How a WFQ scheduler possibly affects the burstiness of a (σ, ρ)-constrained flow is shown in the next example.

Ex. 8: **(output flow constraint for a (σ, ρ)-constrained input flow and WFQ)** *In Fig. 18, the arrival curve α of a (σ, ρ)-constrained flow, the service curve β of a WFQ scheduler with guaranteed rate r and link rate R, and the arrival curve α^* of the resulting outgoing flow determined by eq. (2.11) are shown. In addition, delay and buffer bounds at the WFQ scheduler are drawn where L is the maximum packet length at the access link. The arrival curve α^* of the outgoing flow is a shifted version of the arrival curve α of the incoming flow enforced by the policer. We see that the burstiness σ^* of the outgoing flow is equal to the buffer requirements at the scheduler.*

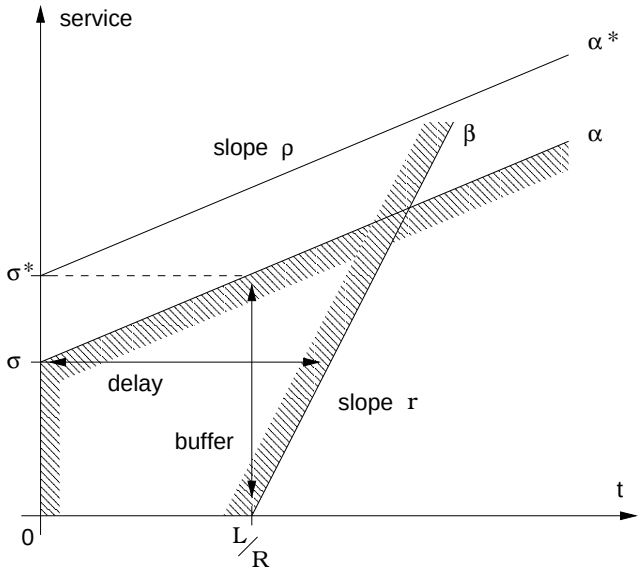

Fig. 18: Output constraint α^* of a (σ, ρ)-constrained flow (arrival curve α) after WFQ scheduling (service curve β) with link rate R, max. packet length L, and guaranteed rate r.

That means, given an allowed burstiness σ^* by the SLA, the burstiness for a (σ, ρ)-constrained flow at the policer can be determined by $\sigma = \sigma^* - \rho \cdot \frac{L}{R}$. Burstiness bounds for a traffic aggregate, meta-classes, and/or distinct flows can be derived from the SLA in this way.

3.1.3 Node architecture

The goal is to compile the functionality of a multi-provider access node into a System on a Chip (SoC) design. We will focus on the packet processing chain in this study. Prefabricated building blocks we can use are different kinds of CPU cores such as ARM or PowerPC cores, on-chip memory blocks of different technology (SDRAM, SRAM), and application-specific memory controllers for handling on-chip or off-chip memory. Configurable parameters of the CPU cores are the technology – and thus the clock frequency – the number and type of functional units, the size and organization of the cache, and the kind and number of special peripheral modules. The size, the technology, the organization, and the width of the memory bus are parameters of the building blocks for RAMs. Building blocks for memory controllers may show different levels of refinement. How selected operating modes of a controller affect the overall memory performance will be discussed in Chapter 4. Particular functions may be accelerated by mapping their functionality onto hardware, e.g. in an FPGA or hardwired.

Since we want to map the packet processing chain onto CPU cores, one possible solution would be to map distinct processing stages to different CPU cores. The CPUs would form a kind of pipeline of processing elements. A packet would be handed from core to core to be entirely processed. Each task needs some memory space to store local context information. A router needs

its routing table, a queue manager keeps track of the FIFO queues of each flow, etc. This data could locally be stored at each CPU or in a global RAM. Finally, all tasks might simply be mapped onto a single CPU. In the same way, special purpose hardware may be assigned to each distinct CPU or offered centrally to all tasks. Some possible configurations are shown in Fig. 19. An additional

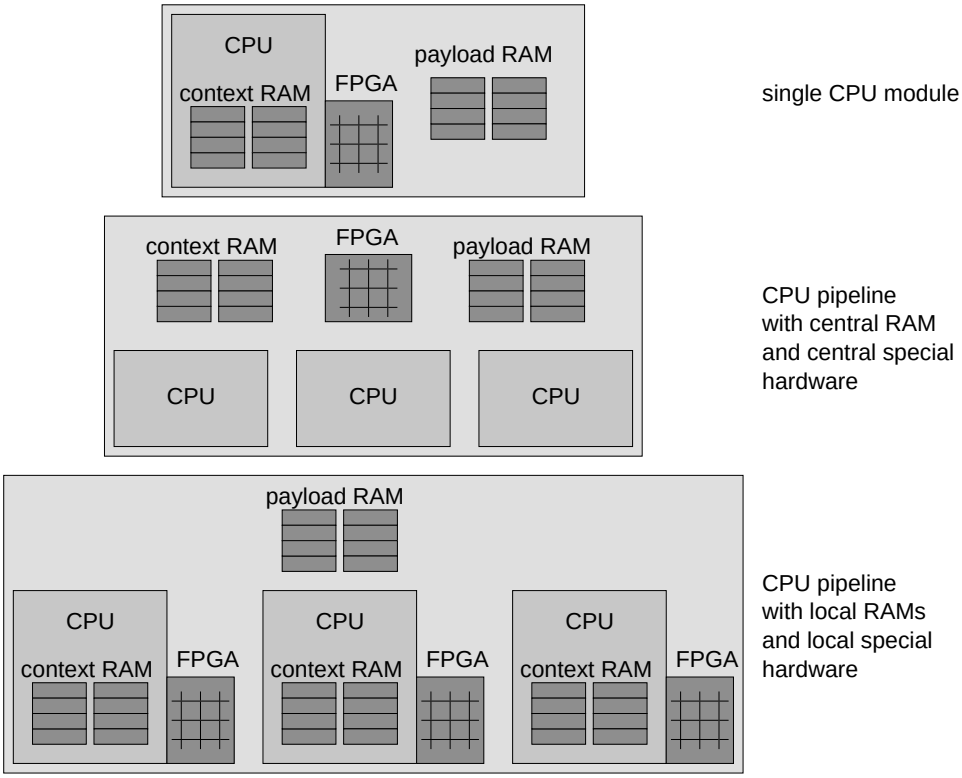

Fig. 19: Different options for the hardware configuration of the multi-provider access node.

RAM is always assumed to store payload. The payload RAM is only accessed once for each packet at the beginning and at the end of the processing chain. In Section 3.3 at the evaluation we start with a pure software implementation on a single CPU and then extend to several CPUs and special hardware blocks.

3.2 Evaluation models

Algorithm and hardware blocks as well as the traffic traces are discussed which are used for the evaluation of options for QoS preservation in multi-service access networks. In detail, three different kinds of models are required:

- *Algorithm models:* Models of this class reproduce the behavior of algorithms for packet processing tasks. The behavior is not bounded by assuming any properties of computing resources. However, the models enable the simultaneous

exploration of algorithms and hardware architectures by generating statistics about their inner state that are used by architecture models to estimate the load of computing resources.

- *Architecture models:* These models imitate the timing behavior of hardware building blocks which can be used to implement a network processor for multi-service access networks. The timing together with the statistics generated by algorithm models are used to estimate the load of hardware resources.

- *Traffic generation models:* Typical network traffic must be modeled to stimulate the network processor. The inter-arrival time of packets determines the frequency of packet processing events. The length of the packet and other packet header information decide the QoS a packet will receive.

The algorithms described in this section are based on the packet processing tasks explained in the preceding chapter. However, due to our new service scheme of green and yellow service, most of the algorithms must be adapted to suit the requirements of multi-service access networks. The architecture models include models for CPUs, RAMs, and some special building blocks. The assumed timing parameters for synchronous DRAMs are derived from the evaluation of memory controller operation modes in Chapter 4. Memory accesses are considered in a more abstract manner than in Chapter 4 by aggregating activation and precharge delays into some few average delays. By the interplay of algorithm and architecture models a more flexible and exhaustive design space exploration is enabled than it has been carried out up to now based on static complexity analysis or existing hardware.

3.2.1 Performance models of algorithms

The algorithms for policing, queuing, and packet scheduling that are used for the evaluation are explained in the following. It is assumed that packets have passed a header parser as well as filter, forwarding, and classifier stages when they enter the policer. These stages are not modeled for the evaluation since their outcome – header fields, next hop address/link, and a QoS class identifier – is constant independently of the chosen algorithms. These stages thus do not affect the QoS preservation behavior of the packet processor in terms of packet delay and buffer space. Moreover, they are candidates for special hardware blocks because header parsing, filtering, classification, and routing must be performed for almost all incoming packets. Since the packet's QoS context information is only available after the classification step, queuing is prohibitive before the classification and the classifier should therefore work at wire speed. In an access network environment, routing tables should be relatively small – at most some thousand entries as in enterprise networks – and the classifier should only manage some tens to one hundred different rules. A hardware implementation is hence moderately straightforward and a reasonable solution.

The support for the suggested multi-service networks with green and yellow QoS requires some adaptations of the policing and queuing components. Exist-

ing solutions for the link scheduler need not be adjusted. The state variables and parameters are mentioned in detail since this information will be used to size up the local memory requirements of each processing stage. The simulation models of all algorithms generate additional information about their inner state and the operations performed during run-time. These statistics are used by architecture models to determine the load of hardware resources. The type of the statistical data issued by the models is described at the end of this subsection.

3.2.1.1 Policer

Since traffic profiles are described by (σ, ρ) specifications, nested token buckets are the only policing element which is considered in the evaluation. An individual policing component covers the check of a single meta-class' traffic profile and all its included green traffic profiles. It is assumed that packets are time-stamped with their arrival time at the packet processor and that they have passed a classification element so that a QoS class identifier including the corresponding meta-class identifier is assigned to each packet.

Compared with common implementations of nested token buckets [81], there is a new additional constraint considering multi-service access networks with our hierarchic service allocation scheme. Since tokens must be taken from both buckets – a flow's green service bucket and the corresponding meta-class bucket – for green packets, a certain amount of tokens in the bucket of the meta-class must be reserved which may only be consumed by green packets. Otherwise, greedy flows within the same meta-class could prevent the generation of green marked packets of other flows by emptying the shared bucket of the meta-class. That is, the condition under which a packet is marked yellow is more precisely stated by saying that the number of tokens in the bucket of the meta-class minus a threshold – the sum of the capacities of the green service buckets in the meta-class – must be greater than the length of the packet.

Alg. 2: (Nested token bucket policer for green profiles and a single yellow profile)

System state parameters:		
N_m		*number of flows within meta-class* m
$(\sigma_{g,i}, \rho_{g,i}), 1 \leq i \leq N_m$	$[Bit, \frac{Bit}{s}]$	*green traffic profiles in meta-class*
(σ_y, ρ_y)	$[Bit, \frac{Bit}{s}]$	*yellow traffic profile for meta-class*
$\sigma_g = \sum_{i=1}^{N_m} \sigma_{g,i}$	$[Bit]$	*sum of reserved green burstiness*
System state variables:		
$t_{LastUpd,i}, 1 \leq i \leq N_m$	$[s]$	*point of time at which token bucket of class* i *has last been updated*
$t_{LastUpd,y}$	$[s]$	*point of time of last update of token bucket for meta-class*
$b_{g,i}, 1 \leq i \leq N_m$	$[Bit]$	*bucket level for green profile* i
b_y	$[Bit]$	*bucket level for meta-class profile*
Input parameters:		
ct	$[s]$	*current time*
l_i	$[Bit]$	*length of packet from flow* i

Initialization *at time* ct

$$\begin{array}{lcl} b_{g,i} & := & \sigma_{g,i}, \quad 1 \leq i \leq N_m \\ b_y & := & \sigma_y \\ t_{LastUpd,i} & := & ct, \quad 1 \leq i \leq N_m \\ t_{LastUpd,y} & := & ct \end{array}$$

At each arrival *of a class* i*'s packet at time* ct *at the policer*

- Update bucket levels:

$$\begin{array}{lcl} b_{g,i} & := & \min\{\sigma_{g,i}, b_{g,i} + \rho_{g,i} \cdot (ct - t_{LastUpd,i})\} \\ b_y & := & \min\{\sigma_y, b_y + \rho_y \cdot (ct - t_{LastUpd,y})\} \\ t_{LastUpd,i} & := & ct \\ t_{LastUpd,y} & := & ct \end{array}$$

- Mark packet and adjust bucket levels accordingly:

`if (` $b_{g,i} - l_i \geq 0$ `){`
    mark packet as green
    $b_{g,i} := b_{g,i} - l_i$
    $b_y := b_y - l_i$
`} else if(` $b_y - \sigma_g - l_i \geq 0$ `){`
    mark packet as yellow
    $b_y := b_y - l_i$
`}`
`else {` *mark packet as red* `}`

3.2.1.2 Link Scheduler

After having marked a packet, the policer hands the packet to the link scheduler. Depending on the state of the link, the scheduler decides whether the packet can immediately be forwarded to the link or whether it must be enqueued. In the latter case, the packet will directly be stored in the priority queue within the scheduler if the packet represents the head-of-line element of the corresponding flow since the flow was not backlogged before. Otherwise, the packet is passed on to the queue manager. As soon as the link becomes idle, the scheduler chooses the packet with the minimum virtual service tag to be transmitted through the link. If there are more packets in the queue for the corresponding flow, a new head-of-line packet will be transferred from the queue manager to the scheduler. A packet's virtual service finish time is generated when the packet enters the priority queue and not already at the packet arrival at the system. In this manner, the queue manager will be prevented from generating large holes in the virtual service time of a flow if the QM is forced to drop packets.

Four packet schedulers are chosen for the evaluation: Deficit Round-Robin (DRR [142]), Self-Clocked Fair Queueing (SCFQ [62]), Starting Potential-based Fair Queueing (SPFQ [150]), and Minimum Delay Self-Clocked Fair Queueing (MD-SCFQ [31]). Since DRR has become a well established basis for comparing scheduling properties, it is included in the comparison although it is

not based on WFQ or Earliest Deadline First (EDF) and thus provides relatively bad worst-case delay bounds. DRR is considered to have low complexity and to supply a fair distribution of bandwidth in the long-term at the expense of relatively poor delay bounds. SCFQ is taken in the evaluation since SCFQ is based on a simple approximation of the virtual service of an underlying fluid server. SCFQ is known to show very good short-term fairness but to provide worst-case delay bounds that depend on other flows sharing the link. Since we are especially interested in tight worst-case delay bounds for green service and tolerate short-term unfairness to some extent, we have also chosen two representatives of the class of rate-proportional servers [152] – SPFQ and MD-SCFQ – that are therefore able to guarantee the best worst-case delay bounds. One could argue that SPFQ needs two priority queues and therefore is rather complex. However, opposed to other algorithms that also provide tight delay bounds and use two priority queues such as WF^{2}Q+ ([153]) and Leap Forward Virtual Clock (LFVC) ([155]), the number of priority queue operations in the packet system can strictly be bounded for SPFQ to a single enqueue or dequeue operation per priority queue and event. With the help of the second priority queue the approximation of the virtual service in the fluid system becomes simple. Contrary to that, MD-SCFQ uses a more complicated calculation of the virtual service but does not need a second priority queue.

We restrict the description of the scheduling algorithms to pointing out the parameters and variables and refer to the corresponding papers for a detailed description of the algorithms. The algorithm for SPFQ is given as an example to show the interplay with the queue manager.

Alg. 3: (WFQ-based scheduling: SCFQ [62], SPFQ [150], MD-SCFQ [31])

System state parameters:		
$N_M = \sum_{m=1}^{M} N_m$		*number of flows in all meta-classes* m
$\Phi_i, 1 \leq i \leq N_M$		*WFQ weights*
MD-SCFQ/SPFQ-specific:		
R	$[Bit/s]$	*link rate*
System state variables:		
V	$[Bit]$	*global virtual time*
$F_i, 1 \leq i \leq N_M$	$[Bit]$	*virtual service finish times*
q_{prio}		*priority queue data structure with up to* N_M *head-of-line elements, sorted according to* F_i*'s*
MD-SCFQ/SPFQ-specific:		
V	$[Bit]$	*global virtual time* $\simeq$ *system potential*
$t_{LastUpd}$	$[s]$	*time of last update of* V
MD-SCFQ-specific:		
l_{all}	$[Bit]$	*sum of the lengths of the head-of-line packets*
Φ_{all}		*sum of the WFQ weights of backlogged flows*

F_{all}	$[Bit]$	*weighted sum of the virtual service finish times of the head-of-line packets*
SPFQ-specific:		
$S_i, 1 \leq i \leq N_M$	$[Bit]$	*virtual service start times*
q^s_{prio}		*priority queue with up to N_M head-of-line elements, sorted according to S_i's*
Input parameters:		
l_i	$[Bit]$	*length of the packet from flow i*
MD-SCFQ/SPFQ-specific:		
ct	$[s]$	*point of time of current event*

Starting Potential-based Fair Queueing (SPFQ):
At each packet arrival *at time ct*

- Update potential:

```
if ( scheduler idle ) { V := 0 , F_i := 0  ∀i }
else { V := V + (ct − t_LastUpd) · R }
t_LastUpd := ct
```

- Packet handling:

```
if ( q_prio contains no entry for flow i ) {
    S_i := max{F_i, V}
    F_i := S_i + l_i/Φ_i      (see eq. (2.3))
    if ( link is idle ) { forward packet to link }
    else { enqueue packet in q_prio and q^s_prio }
}
else { hand packet to queue manager }
```

At each packet departure *(completed service) at time ct*

- Update potential:

$$V := V + (ct - t_{LastUpd}) \cdot R$$
$$t_{LastUpd} := ct$$

- Recalibrate potential:

$$V := \max\{V, \textit{min. start time in } q^s_{prio}\}$$

- Choose next packet for transmission:

```
if ( q_prio is not empty ) {
    dequeue packet with smallest finish time F_i from q_prio
    dequeue corresponding entry of flow i from q^s_prio
    forward packet to link
    if ( flow i is backlogged in the queue manager ) {
        dequeue flow i's head-of-line element from queue manager
        S_i := max{F_i, V}
        F_i := S_i + l_i/Φ_i      (see eq. (2.3))
```

enqueue packet in q_{prio} and q^s_{prio}
}
} else { *scheduler is idle* }

Alg. 4: (Deficit Round-Robin (DRR [142]) packet scheduling)

System state parameters:		
$N_M = \sum_{m=1}^{M} N_m$		*number of flows in all meta-classes* m
R	$[Bit/s]$	*link rate*
$q_i, 1 \leq i \leq N_M$	$[Bit]$	*quantum values of the Round-Robin frame*
System state variables:		
$qc_i, 1 \leq i \leq N_M$	$[Bit]$	*currently accumulated (unused) quantum*
act		*linked list of backlogged flows*
Input parameters:		
l_i	$[Bit]$	*length of the packet from flow* i

3.2.1.3 Queue manager

The queue manager (QM) exchanges packets with the link scheduler component. Since a WFQ-based scheduler relies on per-flow queuing, a QM is obliged to maintain per-flow state information and a distinct FIFO-organized queue per flow must be maintained. In times of congestion a QM for multi-service access networks must warrant that only yellow-marked packets suffer from loss but no green-marked packets. Green and yellow packets of a flow are enqueued in order within the same queue. One may think of different mechanisms to find a yellow packet to push out. Since fairness is not mandatory for yellow service one can simply aggregate pointers to yellow packets from all flows and meta-classes into a single FIFO-organized yellow queue. A yellow packet is either served by the scheduler from a flow's queue or dropped by the queue manager from the yellow queue according to some rule. The corresponding entry in the other queue which is linked with the current entry must then be dropped as well, see Fig. 20. This is our straightforward extension to existing per-flow queuing schemes to cope with graceful service degradation and in-order delivery provided by multi-service access networks.

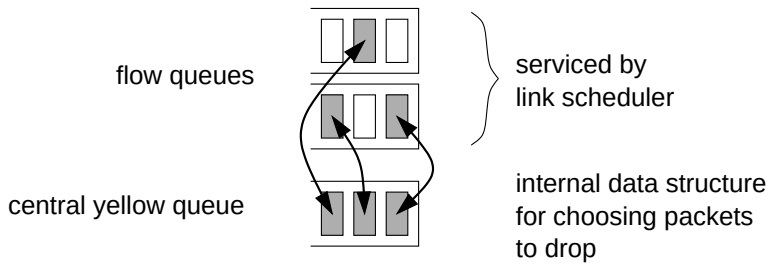

Fig. 20: Administration of yellow-marked traffic by the queue manager.

The payload is stored separately from the QM's parameter memory so that an entry in a FIFO queue only carries a pointer to the packet contents. It is assumed that the QM's local memory is large enough so that the QM's behavior is only determined by the current level of the payload memory.

In times of congestion packets are simply dropped from the tail of the central yellow queue. Since only non-reactive flows are modeled, it virtually makes no difference whether packets are dropped from the front or from the tail of the yellow queue[1]. Opposed to a central yellow queue one may want to implement a QM that provides fair dropping of yellow packets in times of congestion. In this case, a distinct yellow queue is assigned to each flow queue and packets are dropped from the relatively longest yellow queue according to some measure. Suitable measures could be, for instance, the absolute length of a queue or the relative excess of yellow traffic compared with the green reservation. In the latter case, the Longest Queue Drop (LQD [156]) scheme is adapted to the needs of multi-service access networks. Note that we focus on congestion recovery by pushing out packets. Congestion avoidance by preventing packets from enqueuing is already performed by the policer to some extent. We however also look at the common congestion avoidance mechanism RED [54] that we apply to a central yellow queue. Recall that the usual scope of RED assumes reactive flows. Thus, RED cannot completely avoid congestion with our settings, but we can still use our results to estimate the expense of using such a tool.

The following four QMs rely on conventional per-flow queuing for green traffic but are augmented with additional data structures to manage yellow traffic. A QM which employs tail-drop on a central yellow queue (*CYQ*), a slightly modified version of the first QM that also considers the green reservation (*CYQ-enh.*), a QM that applies RED to a central yellow queue (*CYQ-RED*), and a QM that uses tail-drop from the relatively longest queue (*YQ-Fair*) are presented.

Alg. 5: (*CYQ:* QM with central yellow queue and tail-drop)

System state parameters:		
$N_M = \sum_{m=1}^{M} N_m$		*number of flows in all meta-classes* m
$ShrdMem_{max}$	$[Bit]$	*maximum amount of shared memory available for payload*
$Mem_{g,i}, 1 \leq i \leq N_M$	$[Bit]$	*reserved amount of shared memory for flow* i*'s green traffic*
System state variables:		
$ShrdMem$	$[Bit]$	*current amount of shared memory occupied by payload*
$q_i, 1 \leq i \leq N_M$		*data structures to manage flow queues*
q_y		*data structure to manage yellow queue*

[1]For a discussion of the influence of drop-from-front and drop-from-tail on TCP see [100].

Input parameters:		
i		*identifier of the flow to process*
l_i	$[Bit]$	*length of the current packet*

At each packet enqueue event *signalled by the link scheduler*

```
if ( ShrdMem + l_i ≤ ShrdMem_max ) {
    add packet to q_i and – if packet is yellow – to q_y
    ShrdMem := ShrdMem + l_i
} else if ( packet is yellow-marked ) {
        drop packet
} else {
    tail-drop packets from q_y until (ShrdMem + l_i ≤ ShrdMem_max)
    add packet to q_i
    ShrdMem := ShrdMem + l_i
}
```

At each packet dequeue event *signalled by the link scheduler*

dequeue first packet from q_i and – if packet is yellow –
the corresponding entry from q_y
$ShrdMem := ShrdMem - l_i$

The description of the QMs is continued by only outlining the differences to the first QM. The following QM is a slightly modified variant in which it will be tried first to push out yellow packets from the packet's flow queue if the flow occupies more than its minimal buffer reservation. Otherwise, packets are tail-dropped from the central yellow queue.

Alg. 6: (*CYQ-enh.:* Enhanced QM with central yellow queue and tail-drop)

System state variables:		
$ShrdMem_i, 1 \leq i \leq N_M$	$[Bit]$	*current amount of shared memory occupied by payload of flow i*

At each packet enqueue event *signalled by the link scheduler*

```
if ( ShrdMem + l_i ≤ ShrdMem_max ) {
    add packet to q_i and – if packet is yellow – to q_y
    ShrdMem := ShrdMem + l_i, ShrdMem_i := ShrdMem_i + l_i
} else if ( ShrdMem_i + l_i > Mem_g,i ){
   if ( packet is yellow-marked ) {
       drop packet
   } else {
      do {
```

search a yellow packet in q_i beginning from the tail
drop the found yellow packet from q_i and q_y
} while ($ShrdMem + l_i > ShrdMem_{max}$)
add packet to q_i
$ShrdMem := ShrdMem + l_i, ShrdMem_i := ShrdMem_i + l_i$
}
} else {
tail-drop packets from q_y until $(ShrdMem + l_i \leq ShrdMem_{max})$
add packet to q_i and – if packet is yellow – also to q_y
$ShrdMem := ShrdMem + l_i, ShrdMem_i := ShrdMem_i + l_i$
}

The third QM adds the RED [54] congestion avoidance scheme to the central yellow queue of the preceding CYQ-enh. QM. RED defines two thresholds on the average length of the yellow queue. When the first threshold is passed, incoming yellow packets are dropped with a probability that increases linearly with the average length from zero up to a defined level. If the average length reaches the second threshold, all incoming yellow packets must be dropped. In case of congestion, the preceding CYQ-enh. is applied.

Alg. 7: (*CYQ-RED:* CYQ-enh. with RED congestion avoidance for yellow traffic)

System state variables:		
$ShrdMem_y$	$[Bit]$	*current amount of shared memory occupied by the central yellow queue*
avg	$[Bit]$	*average length of the central yellow queue*
Th_{min}	$[Bit]$	*minimum threshold on avg*
Th_{max}	$[Bit]$	*maximum threshold on avg*
p_{drop}		*drop probability*

***At each packet enqueue event** signalled by the link scheduler*

if (*packet is green-marked*) { *continue with* CYQ-enh. }
else {
estimate average queue length avg of yellow queue
if (($Th_{min} \leq avg$) && ($avg < Th_{max}$)) {
calculate drop probability p_{drop}
drop packet with probability p_{drop}
} else if($Th_{max} \leq avg$){ *drop packet* }
if (*packet not dropped*){ *continue with* CYQ-enh. }
}

The last QM is regarded to be fair since a distinct yellow queue is assigned to each flow and packets are dropped from the yellow queue with the highest relative overload. The overload is calculated by the ratio of the current length of a flow's queue to the size of the reserved buffer for green traffic.

Alg. 8: (*YQ-Fair:* Fair QM with per-flow yellow queues)

System state variables:	
$Ovld_i = ShrdMem_i / Mem_{g,i}$	*current overload of flow* $i, 1 \leq i \leq N_M$
$q_{y,i}, 1 \leq i \leq N_M$	*data structures to manage yellow queues for all flows* i

At each packet enqueue event *signalled by the link scheduler*

```
if ( ShrdMem + l_i ≤ ShrdMem_max ){
    add packet to q_i and – if packet is yellow – to q_{y,i}
    ShrdMem := ShrdMem + l_i, ShrdMem_i := ShrdMem_i + l_i
    recalculate Ovld_i
} else if ( packet is yellow-marked ){
        drop current packet
} else {
   do {
        drop packet from yellow queue with highest Ovld value
        recalculate Ovld for that flow
    } while ( ShrdMem + l_i > ShrdMem_max )
    add packet to q_i
    ShrdMem := ShrdMem + l_i, ShrdMem_i := ShrdMem_i + l_i
}
```

3.2.1.4 Statistics generation by algorithm models

The algorithm models generate additional statistics which are handed to architecture models to calculate the load of hardware resources. In this way, the simultaneous exploration of algorithm behavior, timing, and hardware usage is enabled. The information output by the algorithm models can be divided into two main areas: information about the operation performed and information that helps to assess the QoS properties of the system, i.e., the inner state. For the latter purpose, the following data is generated:

- *Queue lengths:* The queue manager can output the lengths of the flow queues as well as the amount of queued yellow traffic.
- *Dropped packets:* Policer and queue manager may output dropped packets.
- *Bucket levels:* The policer may output the current token bucket levels.
- *Virtual time evolution:* The packet scheduler may output the current global virtual time as well as the virtual finish times of all packets in the priority queue.
- *Delay due to queuing and scheduling:* Two points of time are assigned to each packet, the arrival time at the system and the point of time at which the packet has completely been serviced.

In addition, the models generate histograms about the performed operations to process a packet. The counted operations can be of fine granularity such as additions and subtractions or of coarse granularity such as priority queue operations. The histograms of fine-grained operations can directly be used by architecture blocks to assess the load of a hardware component whereas the data about coarse-grained operations must further be processed by intermediate models. The architecture models that analyze these histograms are described in the next subsection. The histograms are output for every elementary packet processing step such as the policing of a packet, enqueuing, dequeuing, etc. The operations counted by the algorithm models are the following:

- *CPU-like:* Arithmetic operations with two inputs; integer and floating-point operations are counted separately.
  - *Min / Max / Cmp:* Minimum, maximum, and compare computations.
  - *Add / Sub:* Addition and subtraction computations.
  - *Mul:* Multiplications.
  - *Div:* Divisions.
- *Other CPU-like operations:*
  - *Branch*
  - *Register copy:* Elementary register copy operations, e.g., for initializing a variable, which are not counted otherwise as they are not performed together with any arithmetic operation.
  - *Address offset calculations:* Operation needed to address particular fields in a structured data element whereas the index is only known during runtime. Since there are architectures that provide execution units just for this operation type, it is counted separately from arithmetic operations.
- *Memory accesses:* Accesses of the context information, such as the state parameters and variables including the packet payload, are counted. It is assumed that the context information is stored in on-chip or off-chip RAM. The type of the access – read or write – and the length of the access in Bit are recorded.
- *Priority queue operations:* Priority queue operations are: enqueue, dequeue, read minimum, delete. The operations are recorded together with the current length of the priority queue. These histograms must be processed by further architecture blocks to assess an implementation of a sorted data structure.
- *Dynamic memory allocation:* The queue manager relies on dynamic memory allocation. The number of memory allocation and deallocation events is counted. These values must be handled by further architecture models to consider an implementation of this task.

Counting methodology
An algorithm is presented that is used to analyse packet processing tasks off-line. Each so-called basic block of the description of a packet processing task is annotated with the high- and low-level operations that must be performed by network processing hardware to execute the corresponding code fragment. In this way, overall operation histograms can be generated during the (simulated) run-time of the system.
Assumptions and limitations: In order to generate the operation histograms at the high level of a given performance model of an algorithm, some assumptions must be made:

- Memory accesses caused by instruction fetches cannot be taken into account. Thus, if a task is implemented in software, it will be assumed that the program entirely fits into the instruction cache.

- A set of free-programmable registers is available to hold interim results within a basic block of code and to hand interim results from basic block to basic block.[2]

- Read accesses to context information always generate RAM accesses when they appear first during an elementary packet processing task. Write accesses to context information always generate RAM accesses. That means, data cache misses will always be assumed if the algorithms are mapped onto a CPU.

- Procedure calls cannot be taken into account. That means in case of a software implementation, the elementary packet processing tasks are implemented without any call hierarchy, e.g., by using in-line functions.

- When counting CPU instructions it is assumed that the CPU supports register-direct, register-indirect, absolute, and indexed (register-indirect plus an offset) addressing modes. The first three addressing modes do not generate any additional overhead to calculate the effective address. Only indexed addressing will require an address offset calculation if the offset is only known during run-time.

- An algorithm model may not adequately reflect reasonable data structures for an implementation. The RAM access and CPU operation counters must however reproduce the statistics for a virtual implementation. That is, the operations to count may not directly be seen in the performance model and require further assumptions about the data structures to use in an implementation.

Before the operations' counting algorithm can be presented, two further definitions are required that determine our region of interest – a basic block of code – and shared data between these regions.

Def. 16: **(Basic block)** *A* basic block *of code is a code fragment given in a programming language where the control flow is only allowed to enter the sequence of*

[2]Note: The application of the following Alg. 9 on our algorithm models has shown that 16 registers are sufficient for the configurations presented in this chapter.

instructions at the beginning of the fragment and to branch at the end of the fragment but not in between. That is, once the control flow enters a basic block, the instructions of the block are deterministically processed in order.

Def. 17: **(Active variable)** *An* active variable *is a data item which is handed from a basic block to another basic block and read in the latter block – before it may be possibly written. That is, an active variable is a candidate to be held by a temporary register to save memory accesses.*

Alg. 9: **(Off-line counting of operations)** *For a given elementary packet processing task (dequeue, enqueue, policing, etc.) specified by a programming language:*

- *Detect the basic blocks.*
- *Determine the control flow between the basic blocks.*
- *Beginning from the entry point of the control flow of the overall task, determine the sets of active variables at the entry point and at the branch point of every basic block.*
- *For every basic block:*
  - *Extract code dealing with priority queues and dynamic memory management. Count priority queue and dynamic memory allocation operations.*
  - *In the remaining code fragment:*
    - *Detect variables and constants which belong to the context information and which are not active.*
    - *Count the required memory accesses and address offset calculations to read these variables from memory.*
    - *Count the required CPU operations.*
    - *Detect the context variables which have been set in the basic block.*
    - *Count the required write accesses and address offset calculations to write these variables back to memory at the end of the basic block.*
  - *Assign the determined counter values to the basic block.*

In the end, overall histograms of operations can be gathered for arbitrary orders of packet processing events with the help of the assigned counter values per basic block. Operations will only be counted if the actual control flow passes the corresponding basic block during run-time of the simulation.

Ex. 9: **(Off-line counting of operations)** *The following code fragment is taken from an update method of a potential-based packet scheduler. Context variables are* $lastUpdate$, $linkrate$, *and* V. *At the entry point of the basic block,* $time$ *is active. At the end of the basic block,* $lastUpdate$ *must be active. The write and read counter methods for the memory accesses have the length of the access and the number of accesses as parameters.*

```
[...]
else if (!HPTrafficStall){
  // -- basic block starts here --

  V += (time - lastUpdate) * linkrate;
  lastUpdate = time;

  // statistics ------
  cnt.addRead(Prec.TIME,1)      // lastUpdate
  cnt.addRead(Prec.VTIME,1);    // V
  cnt.addRead(Prec.LINKRATE,1); // linkrate
  cnt.addSubFP += 2;            // floating-point add/sub
  cnt.mulFP++;                  // floating-point mul
  cnt.copy++;                   // register copy
  cnt.addWrite(Prec.TIME,1);    // lastUpdate
  cnt.addWrite(Prec.VTIME,1);   // V
  // ------
}
// -- basic block ends here --
else {
[...]
```

First, the non-active context variables $lastUpdate$, $linkrate$, and V must be read from memory. Then, the arithmetic operations can be performed. Last, the context variables $lastUpdate$ and V are written back to memory. Note that the context variable $lastUpdate$ is written back to memory and held by a register – this is why an additional register copy operation is counted – because $lastUpdate$ must be active at the end of the basic block.

3.2.2 Architecture models

Histograms of operations and other statistical data are output by algorithm models. This output is analyzed to derive estimations of the utilization of resources by models described in this subsection. They consider the operation-specific timing of hardware blocks. Since these models are simulated together with the algorithm models, the simultaneous exploration of algorithm behavior and resource load is enabled without relying on any execution traces.

3.2.2.1 CPU timing model

As it will be shown by a discussion of related work in Section 3.4, all currently available network processors employ a general-purpose computing core to some extent to flexibly adapt to protocol changes or new communication standards. We also start with the assumption of a software implementation and profile our implementation to point out whether there are subtasks that should be moved to hardware. The CPU cores taken into consideration are listed in this subsection.

Only the latency introduced by a CPU's execution stage of the pipeline is taken into account by a timing model. It is assumed that the latency for other stages like fetch and decode are virtually hidden by the concurrent processing

in the pipeline. If the execution stage allows variable delay for a class of operations, only the maximum value will be considered. The individual values for the operation counts in the histograms generated by the algorithm performance models are weighted with the latency of the execution stage and the resulting values are summed up. Since this kind of analysis is performed in fixed periods, a rough estimate of the average load of the CPU in a particular period can be derived. We do not make use of any superscalar architectural features of the CPU because the histograms do not provide any information about the order of the operations and we cannot perform any dependency analysis.

Three simple CPU timing models have been implemented in order to assess the load generated by packet processing tasks. ARM cores are used for a broad range of embedded systems. In particular, ARM CPUs are employed for systems where the power dissipation is a critical design constraint such as in handheld devices. The application area of the ARM9 core we have modeled ([6]) is focused on integer computation. The ARM9 neither has a floating-point unit nor a divider unit. These operations must be emulated by integer operations. We have applied the methods described in ([82], chapter 4) to map floating-point operations and divisions to integer computations. The Hitachi 7750 ("SH4", [83]) is a more sophisticated embedded CPU with a floating-point unit including a divider. Finally, a PowerPC 750 ([87]) represents the computing power utilized in common personal computer systems. The timing values used for the evaluation are listed in Tab. 2. The timing values for floating-point operations assume a precision of 64 Bit whereas the integer operations use a precision of 32 Bit. Although the assumption of high-precision operations turns out to be rather inappropriate for an efficient implementation of the algorithms, it however causes the calculation of load values which are closer to the worst-case so that the design-space exploration becomes more reliable.

Tab. 2: Timing of the CPU models in clock cycles.

Operation [cycles]		*ARM9ES*	*SH4*	*PPC 750*
Integer	Min / Max / Cmp	1	1	1
	Add / Sub	3	1	1
	Multiply	5	4	6
	Division	224	4	19
	Address offset	3	1	1
Floating-point	Min / Max / Cmp	8	5	3
	Add / Sub	17	9	3
	Multiply	18	9	4
	Division	378	26	31
Branch		3	3	3
Register copy		1	1	1
Clock [MHz]		120	200	500

3.2.2.2 RAM timing model

Two representative timing models for off-chip RAMs are chosen for the evaluation. The first RAM is a common PC100-compliant SDRAM [88] as it will be used for the evaluation of memory controller access schemes in Chapter 4. The timing employed by the SDRAM model is derived by averaging over all accesses of all simulation runs presented in Section 4.4 applying open-page mode with interleaved address mapping but without overlapped processing. SDRAMs are preferred over RDRAMs since accesses of parameters and variables are rather small so that an RDRAM cannot take advantage of its superior bus bandwidth. The second RAM is a pipelined static RAM with late write. This is a typical component used in order to implement off-chip caches. All major RAM manufacturers offer SRAMs of this class.

Since the algorithm models only provide access histograms without any information about the order of accesses, an average latency overhead for reads and writes is considered in the SDRAM case that aggregates the latencies for activations and precharges. For the same reason, we do not make use of the late write facility of the SRAM. The timing used for the evaluation is given in Tab. 3. The histograms generated by the algorithm models distinguish the access type – read or write – and the length of an access. As the histograms are analyzed in regular intervals, one can estimate the load of the RAM within a period by summing up all the delays caused by the accesses in that period divided by the length of the period. The delay d caused by a single access is calculated by

$$d = \left(\left\lceil \frac{\text{access length}}{\text{memory bus width}} \right\rceil + \text{access overhead} \right) \cdot \frac{1}{\text{bus clock}}$$

Tab. 3: Timing of the RAM models.

Parameter		*SDRAM*	*SRAM*
width of memory bus	[Bit]	32	32
clock of the memory bus	[MHz]	100	166
read access overhead	[cycles]	4	2
write access overhead	[cycles]	3	2

3.2.2.3 Priority queue model

Since the complexity for sorting and searching heavily depends on the chosen data structure, the behavior of priority queues has been separated from the algorithm models of the processing chain to ease the analysis of the influence of a priority queue on the overall performance.

A priority queue is a data structure in which elements are sorted in increasing/decreasing order of assigned key values. A priority queue is needed for two tasks of the packet processing chain. The packet scheduler must sort packets according to deadlines or virtual finish times. The fair queue manager drops

packets from the relatively longest queue during times of congestion. Thus, the queue manager maintains a priority queue in which flows are sorted according to their current buffer occupancies.

A priority queue for a link scheduler must support the following operations:

- *Enqueue:* A new entry is sorted into the data structure.
- *Dequeue:* The entry with the minimum/maximum key is removed.

A priority queue for a fair queue manager must additionally support:

- *Read out:* The entry with the maximum key is read out (but not removed).
- *Delete:* An entry with a given field value (which is not the key) is removed.

The right choice for a suited data structure for sorting and searching depends on the number of entries to sort, the type of operations to support, the characteristics of the key distribution, and the frequency of operations. In our case, the maximum number of entries in a priority queue can be bounded since at most one entry per flow must be sorted into the data structure. The maximum number of entries is relatively small since only some hundred flows must at most be scheduled. The support for merge operations is not required because only a single key is processed at a time. The keys may be arbitrary floating-point values which are usually not limited by a fixed range. Due to the small number of flows to sort, a software implementation of a mature data structure is chosen. Thus, the priority queue statistics output by the algorithm models are transformed into CPU and RAM operations that are additionally transferred to the architecture blocks. A heap organized binary tree [96] mapped onto a fixed-sized array is assumed for the data structure implementing a priority queue. A heap shows logarithmic complexity with the number of entries to sort, inherently maintains a balanced tree, and child and parent nodes within the tree can easily be addressed by shift and increment operations if mapped onto an array. Since no additional operations are required to balance the data structure, heaps can be analyzed well. Although the key values cannot be bounded to a fixed range or approximated by integers, a heap nevertheless competes reasonably well with more sophisticated priority queue data structures [133, 93].

3.2.2.4 Dynamic memory allocation model

Dynamic memory allocation is a special processing block which has been separated from the algorithm models to evaluate its impact on the overall system's performance. The queue manager relies on dynamic memory allocation. A packet's payload must be stored at a packet arrival and context information extracted from the packet header must be appended to a flow's FIFO queue. For both tasks, the QM needs to dynamically manage memory space. A simple software implementation with fixed-sized blocks is considered. The memory allocation counters output by the algorithm models are transformed into CPU and RAM operations which are also transferred to the CPU and RAM models.

The concept to store dynamically generated data of variable length as a linked list of fixed-sized segments has been established several decades ago [164]. This mechanism still is the most common solution used in switches and routers since the management of dynamic memory allocation becomes simple due to the absence of fragmentation loss. It therefore is well suited for storage systems which work under real-time constraints and where the length of the data to store may not be known at the beginning of the store process.

The basic idea of our simple software implementation is that a memory region is exclusively reserved for a free list of pointers to available memory blocks. The free list is organized as a stack of pointers. This way, the stack can be easily managed by two additional pointers, e.g. held by registers, if the stack region and the block memory do not overlap. However, some memory space is wasted since the reserved space for the stack cannot be used for storing additional memory blocks although the stack may be empty. Nevertheless, the overhead for reserving a separate memory region for the free list is about 6% by using 68 Byte segments (64 Byte payload plus 4 Byte address to the next segment) and 4 Byte addresses as an example. By applying the results from [164], our memory allocation scheme utilizes the memory best – in terms of the least share of occupied memory for storing pointers in order to link segments – for average packet lengths in the range of 500 to 600 Byte. Fig. 21 shows the memory allocation scheme. As long as $freeListStart$ is not equal to $freeListBase$ there are free memory segments available and we can allocate a memory block where the entry at $freeListStart$ points to. $freeListStart$ is de-

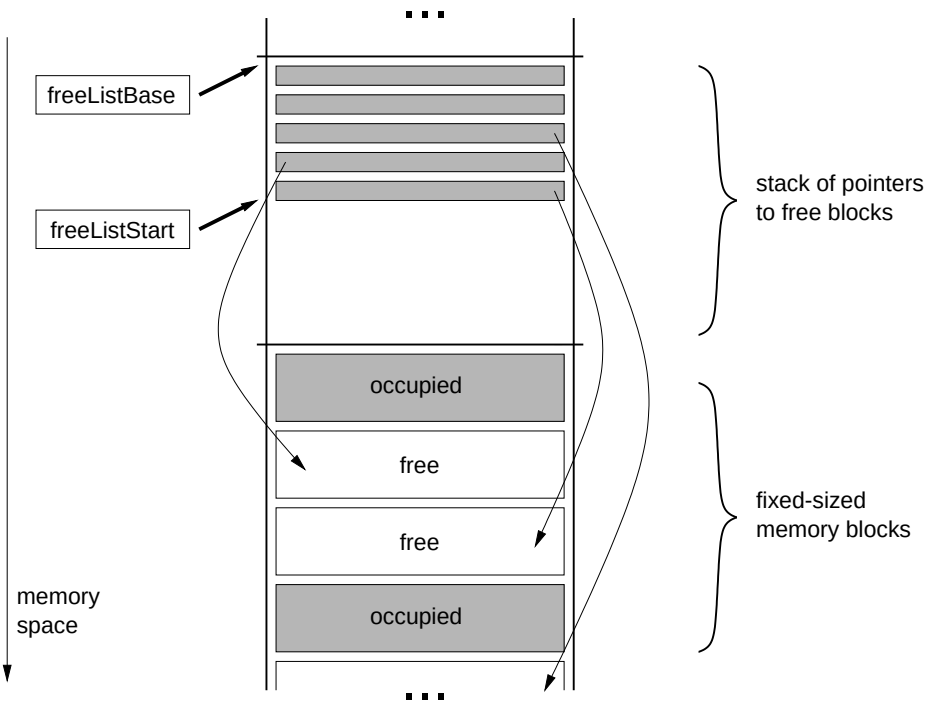

Fig. 21: Simple dynamic memory allocation scheme using exclusive memory areas and fixed-sized memory blocks.

creased afterwards. When a block becomes free again, its address is pushed onto the stack and $freeListStart$ is increased. A packet's payload may be distributed over several memory blocks.

3.2.3 Stimuli

In order to stimulate the packet processing chain, there are different options for choosing traffic patterns. One can use:

- *Public traffic traces from the Internet:* Traces that are publicly available in the Internet usually record traffic seen at backbone routers. In particular, the number of distinct flows is high and understanding their traffic profile needs a thorough analysis of the traces. Traces can be found at various places in the Internet, for instance, [3, 109] are two well-known sites among others.

- *Real traffic sources:* For instance, one could imitate the exact behavior of TCP to initiate real HTTP downloads and use video codecs to generate UDP traffic, etc. That is, the simulated packet processing chain would participate in the network protocol stack of a real system.

- *Statistical source models:* There is a variety of statistical source models that have been derived by measurement and analysis of Internet traffic. The following references should therefore only be seen as starting points for further reading. World Wide Web traffic is modeled in [9, 10]. It is in particular pointed out that the characteristics of WWW traffic change over the years. Studies which especially focus on TCP behavior can be found in [121, 27]. The former work discusses TCP's steady-state behavior whereas the latter paper enhances the model by adding startup effects which are important to imitate short-term TCP connections. Finally, models for video traffic with variable bit rate are presented in [107, 56, 134]. Further studies that focus on traffic characteristics to facilitate traffic engineering can be found in [123, 50].

  In our case where access networks are investigated Internet backbone traces cannot be used effectively. These traces aggregate a high number of flows on a best-effort basis and their time stamps reflect backbone speeds. The integration into a real system would require to emulate the behavior of the access link provider and the contents providers. Finally, statistical source models have been derived by analyzing traffic traces with a duration from hours up to days. However, the periods that we use for the analysis of resource load and the behavior of the packet processing chain only cover at most some minutes so that the statistical models would probably not reflect a representative traffic pattern. It has thus been decided to use a set of traces generated by some own synthetic models and not to cope with existing models, Internet traces, or the overhead of implementing TCP/UDP. The traces should reflect an aggressive load for the packet processing chain. The following types of traces have been generated to model traffic patterns on incoming LAN links, e.g. fast-Ethernet:

- *Constant Bit Rate (CBR) sources:* Every 10 ms, a packet of the size 128 Byte is generated. In this way, a 64 KBit/s uncompressed voice source is modeled.
- *Variable Bit Rate (VBR) Video:* A packet corresponds to a video frame. MPEG or H.263 coded video shows some periodic behavior. Every eight to 12 video frames, an intra-coded frame is transmitted which is relatively large whereas the other frames are predicted by some inter-frame coding and hence are smaller. The inter-arrival time of the packets is determined by the video standard. For instance, NTSC uses an approx. 30 Hz frame rate and PAL 25 Hz respectively. If interlaced frame coding is employed, the inter-arrival time of packets will halve. The inter-arrival time may show some jitter due to variable coding delay. Video traffic resembling 128 KBit/s PAL MPEG, 42 KBit/s NTSC H.263, and an aggregate of two MPEG streams and an H.263 stream are modeled.
- *Call signalling:* Around ten packets are generated every three to four seconds with lengths varying from 128 to 512 Byte and a peak rate of 5 MBit/s to model the connection establishment procedure for voice traffic.
- *HTTP-like traffic:* HTTP-requests are modeled by bursts of five to ten packets with varying size of 40 to 300 Bytes and a peak rate of 5 MBit/s. These bursts appear every 0.5 to five seconds. HTTP-downloads are imitated by bursts of maximum-sized packets (1536 Byte) with a peak rate of approx. 50 MBit/s and lengths of four to 10 ms.
- *FTP-like traffic:* FTP downloads are simply modeled by longer bursts than in the HTTP case. The burst duration now varies from 10 ms to 0.5 s.
- *Transactions:* Every 0.3 to two seconds, two to five packets with varying length from 250 to 320 Byte are generated to imitate transactions with banks, bandwidth brokers, etc. The peak rate is approx. 10 MBit/s.
- *Flooding:* Maximum small packets flood the packet processing chain. The packet size varies from 40 to approx. 46 Bytes and the peak rate from 10 to 50 MBit/s respectively.

A list of the flow characteristics of the synthetic traces is given in Tab. 4. $N(avg, dev)$ stands for a normal distribution with mean avg and standard deviation dev and $U(l, r)$ denotes a uniform distribution in the interval $[l, r)$. Packet lengths are rounded to the next integer. Packet lengths below 40 Byte are rounded up to 40 Byte and lengths above 1536 Byte are round off to 1536 Byte.

It is realistic that all these types of traffic patterns appear from the customer's network to the access network since the access link may be used to interconnect distributed enterprise networks via a VPN. Thus, enterprise servers may also be distributed in the same way. Moreover, the enterprise may offer information for the public Internet. Some of these traces are displayed in Fig. 22 as an example.

Besides the traces that stimulate the system at the inputs a timed link model is required that signals to the system when the link is idle and ready to transmit

Tab. 4: Characteristics of the source models for the synthetic trace generation. After the generation step, the packet lengths below 40 Byte have subsequently been rounded up to 40 Byte and lengths above 1536 Byte have been rounded off to 1536 Byte.

Trace type	*packet inter-arrival time* [ms]	*packet length* [Byte]	*burst length* [ms]	*burst spacing* [s]
CBR voice	10	128	*no bursts*	–
Video P-frame	$N(20, 5)$	$N(230, 50)$	*no bursts*	–
Video I-frame	$N(160, 40)$	$N(900, 100)$	*no bursts*	–
Signalling	$N(0.5, 0.05)$	$U(128, 512)$	$U(4.5, 5.5)$	$U(3, 4)$
HTTP request	$N(0.25, 0.1)$	$N(150, 100)$	$U(1, 3)$	$U(0.5, 5)$
HTTP downld.	$N(0.25, 0.1)$	$N(1700, 400)$	$U(4, 10)$	$U(0.5, 5)$
FTP download	$N(0.25, 0.1)$	$N(1700, 200)$	$U(10, 500)$	$U(0.5, 5)$
Transactions	$N(0.25, 0.1)$	$N(275, 30)$	$U(0.5, 1)$	$U(0.3, 2)$
Flooding 50Mb	$N(6.7 \cdot 10^{-3}, 1 \cdot 10^{-3})$	$N(40, 2)$	*no bursts*	–
Flooding 32Mb	$N(1 \cdot 10^{-2}, 2 \cdot 10^{-3})$	$N(40, 2)$	*no bursts*	–
Flooding 10Mb	$N(3.3 \cdot 10^{-2}, 3 \cdot 10^{-3})$	$N(40, 2)$	*no bursts*	–

a packet as well as when the transmission of a packet ends according to the link bandwidth. In this way, the link model stimulates the packet scheduler to choose packets for transmission.

3.3 Results

The following design space exploration not only shows the influence of combined policing, queuing, and link scheduling on the Quality of Service (QoS) behavior of a network processor for multi-service access networks but also the expense in terms of hardware resources required to implement sophisticated QoS preservation.

3.3.1 Evaluation methodology

The goal of this study is to perform a design space exploration of different combinations of policers, queue managers, and link schedulers stimulated by different traffic traces with varying characteristics. We are interested in the timing and fairness behavior of the algorithms as well as the utilization of hardware resources for every configuration. However, since the number of suitable combinations is only some hundred, the design space exploration is performed by exhaustively simulating every possible combination.

Simulation environment: The MOSES Tool Suite [4, 90] is used as modeling and discrete event simulation environment. MOSES allows to simultaneously use different formalisms with or without graphical notation to model heterogeneous systems. Two graphical notations are used for this study, namely process

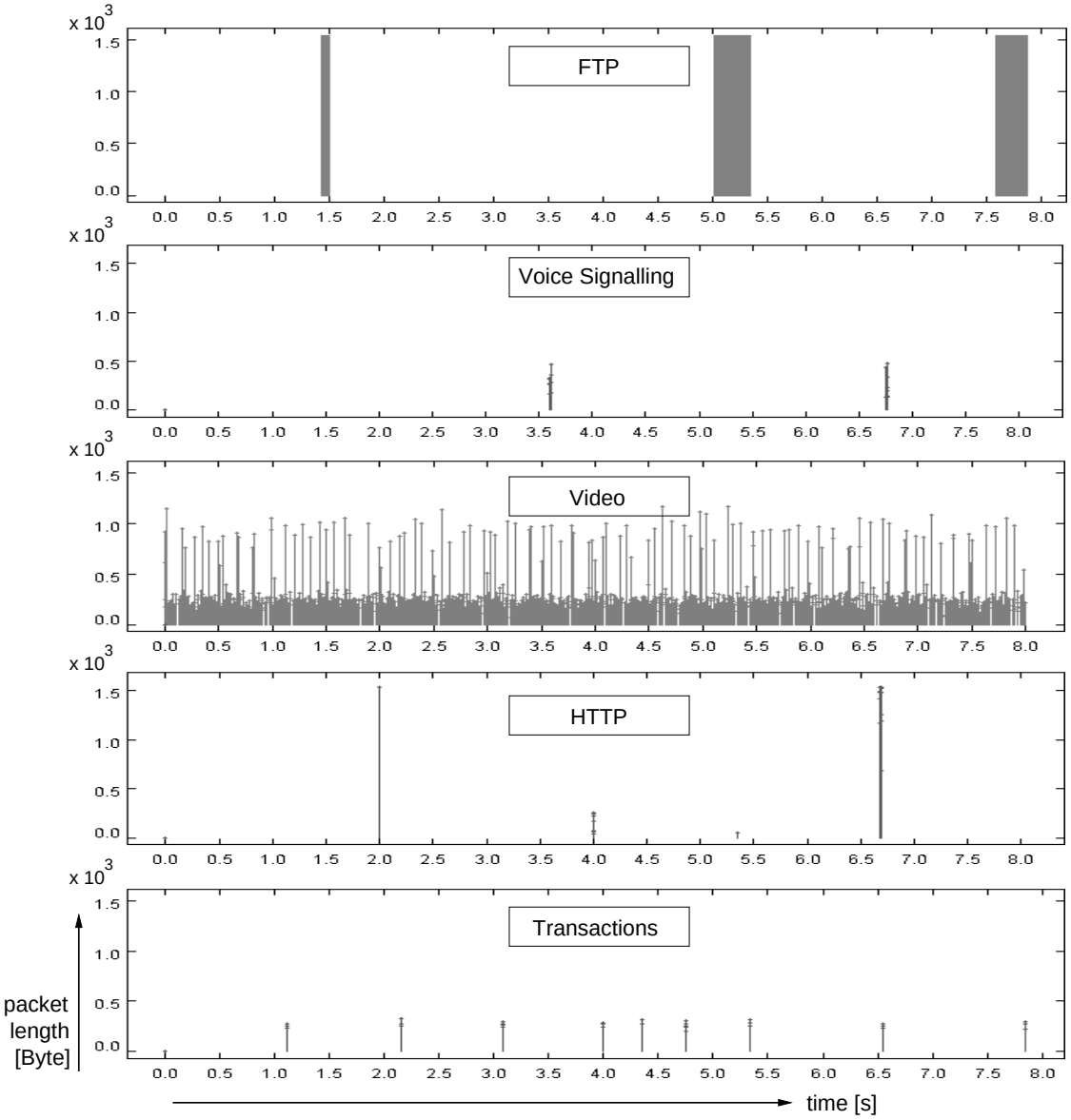

Fig. 22: Some parts of synthetic traffic traces. An excerpt of eight seconds is shown.

networks and Petri nets. Process networks as defined in MOSES consist of a collection of concurrently executing processes which have input and output ports and which are interconnected. The execution of a process is dataflow-driven, that is, as soon as a data token appears at any input port of a process, its execution is altered or initiated. As a result of an execution, data tokens are generated at the output ports. Process networks can be nested hierarchically and the execution model of a process can be specified by any supported formalism. The simulation backplane and process networks in MOSES support a notion of time. Data tokens are therefore assigned time stamps that may affect the execution of a process. Process networks are used to model the interconnection of policer, queue manager, and link scheduler components. Whole packets

or excerpts from the packet header are passed from component to component. The programming language Java is employed for the underlying computational model of a component to execute a process – an algorithm and the generation of statistical data in our study. In addition to that, processes specified by Petri nets are employed to gather statistics or prepare the timing analysis of a configuration.

Determining the load of resources: Since we want to explore the behavior of different combinations of algorithms under ideal conditions, timing information generated by the architecture blocks for CPUs and RAMs are not feed back into the algorithm models. The load of hardware components is calculated in the way sketched in Fig. 23. Different time bases affect the behavior

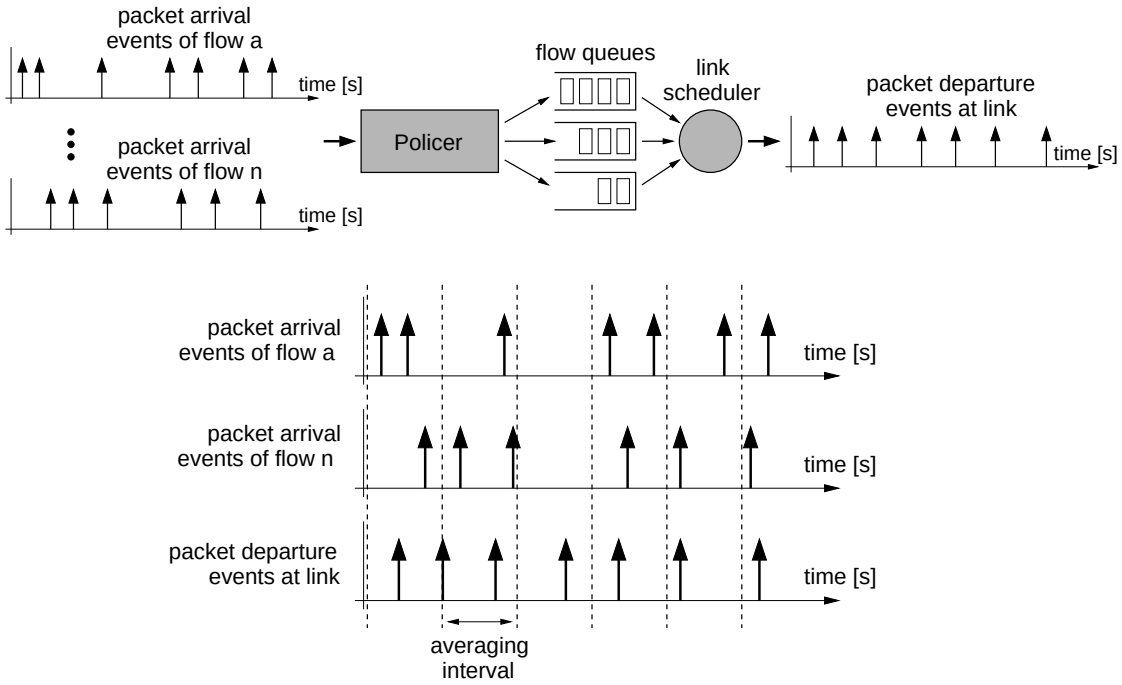

Fig. 23: Methodology for the calculation of load values.

of the simulated system. On the one hand, packet arrival events from different packet traces stimulate the system at the input. On the other hand, packet departure events at the outgoing link trigger further packet dequeuing tasks. With each packet arrival or departure event a number of operation histograms are generated by the policer, the queue manager, and the link scheduler. These histograms are accumulated for the period of a defined averaging interval $avgIntv$. The accumulated histogram is evaluated by architecture blocks. The resulting delay scaled to the length of the averaging interval determines the load of the corresponding architecture block for that averaging interval. This procedure is repeated for every following averaging interval. In this way, the progress of the load of a component will be modeled with an $avgIntv$ granularity if we allow an additional delay of $avgIntv$ to be experienced by the packet processing chain in order to process a packet arrival or departure event. The additional delay is

caused by the buffering of events to balance the load of a hardware component within a period of $avgIntv$.

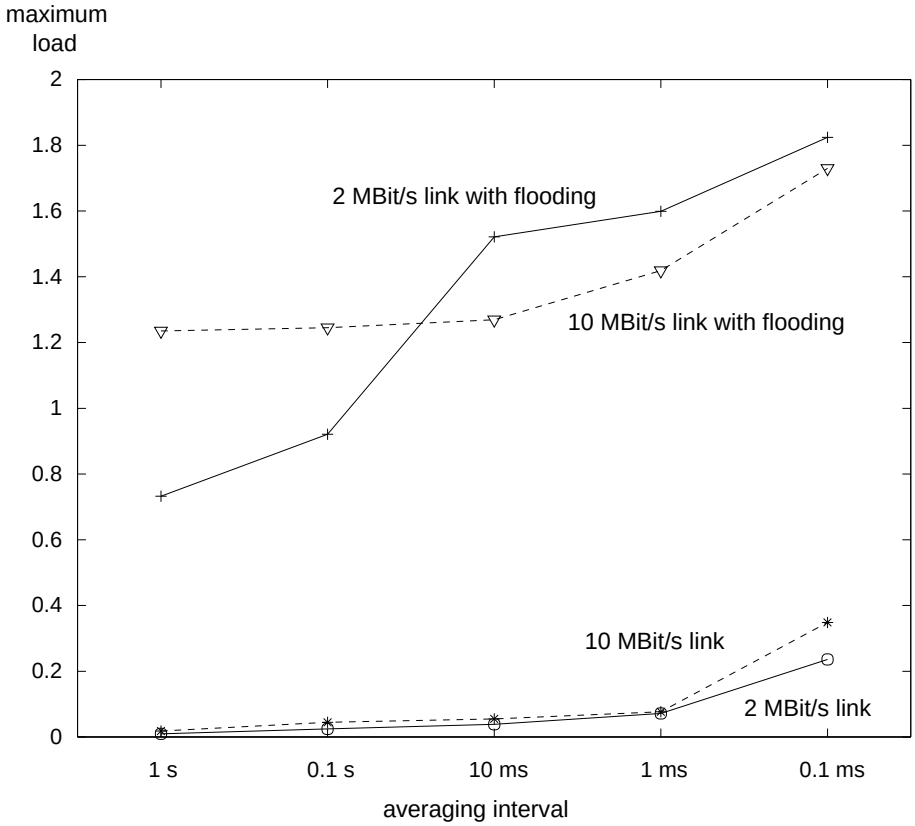

Fig. 24: Impact of the length of the averaging interval on the maximum experienced CPU load.

Impact of buffering packet processing events: In order to estimate the impact of the length of the averaging interval on the load peaks of an architecture block, the duration of $avgIntv$ is varied for some simulation runs of selected configurations. The result is shown in Fig. 24. The maximum load generated with an ARM CPU core is taken by means of an example. Four configurations are investigated. The two lower curves resemble a light load caused by traffic traces with strongly varying characteristics. The two upper curves show the load caused by greedy traffic traces with very small packets. For these particular configurations, the ARM core is overloaded for almost all choices of $avgIntv$. For most of the configurations one will only experience a noticeable increase in the maximum load from the long-term average if $avgIntv$ is set below 10 ms. On the one hand, the maximum experienced load for the light-load configurations may raise by up to a factor of three if $avgIntv$ is reduced from 1 ms to 0.1 ms. On the other hand, the load caused by high-load configurations only increases by 20% for the same change of $avgIntv$. That means, since the greedy traces stress the system for a long period of time, the system cannot significantly reduce its load by longer buffering events before computation. The reverse statement is obviously true for the light-load configurations. For our settings it is not very

suitable to reduce $avgIntv$ far below 0.1 ms since $avgIntv$ would then only be a fraction of the transmission time of an average packet. The following list in Tab. 5 shows some examples of the transmission time of packets through links with different bandwidths. By analyzing the load of hardware resources with a

Tab. 5: Packet transmission time for some link bandwidth - packet length combinations.

packet size	*link bandwidth - packet transmission time*		
$[Byte]$	*2 MBit/s*	*10 MBit/s*	*30 MBit/s*
40	$1.6 \cdot 10^{-4}$s	$3.2 \cdot 10^{-5}$s	$1.1 \cdot 10^{-5}$s
512	$2.1 \cdot 10^{-3}$s	$4.1 \cdot 10^{-4}$s	$1.4 \cdot 10^{-4}$s
1536	$6.2 \cdot 10^{-3}$s	$1.2 \cdot 10^{-3}$s	$4.1 \cdot 10^{-4}$s

too low $avgIntv$, one would very likely see high load peaks followed by periods of idle resources. Four our studies averaging intervals of length 1 ms are used. This way we make a compromise between the additional delay introduced by buffering events before computation and the resolution in order to recognize load peaks. Since link bandwidths up to 10 MBit/s will be investigated in the following experiments, the delay distortion introduced by buffering events before computation is in the same order of magnitude as the delay penalty caused by non-preemptive packet transmission.

3.3.2 Experiments and analysis

The main goal of the evaluation is to answer the following questions:

- How do different combinations of queue managers and packet schedulers perform to preserve the QoS? Do the algorithms protect flows against misbehaving flows in times of congestion? Are the worst-case delay guarantees kept?
- How do CPU load values generated by the different components compare to each other? Does a component – policer, queue manager, or packet scheduler – dominate the others or is the overall load rather balanced among the tasks?
- How does the CPU load of the components scale with a higher link rate? We are interested in future access networks and look at two representative bandwidths of the access link. A 2 MBit/s link imitates forthcoming high-bandwidth cable modem and DSL solutions whereas a 10 MBit/s link resembles, for instance, the capability of future wireless access networks.
- What are worst-case situations for combined policing, queuing, and scheduling? Although it is simple to separately define worst cases for each component, tracking down worst-cases for the overall system is not straightforward.
- How does the behavior of the queue manager impact the fairness properties of the system in times of congestion? Is it worthwhile to afford a per-flow distinction during times of congestion?

- Can we encounter noticeable differences between the behavior of a Deficit Round-Robin (DRR)- and a Weighted Fair Queueing (WFQ)-based scheduler? The system must only cope with a relatively small number of classes at an access network. The well-known behavior of DRR that delay properties decline with an increasing number of flows sharing a link may not be observed with our access link settings.
- How does a queue manager with per-flow state distinction compare with the common Random Early detection (RED) [54] in terms of complexity?
- Can performance bottle-necks be detected or would an unoptimized software implementation already perform well? That is, is there a need to optimize the implementation and move tasks to hardware blocks, etc.?
- How does high-priority voice traffic disturb the system behavior?
- In the end, can a reasonable trade-off be found among the complexity, the fairness, and the delay properties of the overall system?

3.3.2.1 System specifications

The system for the evaluation is displayed in Fig. 25. It corresponds to the structure discussed in Subsection 3.1.2. Four meta-classes are defined containing eight classes all together. The WWW traffic meta-class consists of HTTP and FTP traffic. Transactions are handled by a separate class. Video and voice signalling form their own media meta-class. A Virtual Private Network (VPN) meta-class holds three classes. Voice traffic does not go through the main path but has highest priority under all circumstances guaranteed by a simple static priority scheme. All inputs are stimulated by corresponding packet traces as described beforehand in Tab. 4. Each meta-class is policed by nested token-buckets as depicted in Subsection 3.2.1.1. The queue manager communicates with the scheduler component since the scheduler decides whether a packet can immediately be forwarded to the link or must be buffered. Finally, the model of the outgoing link stimulates the scheduler to choose packets for transmission.

The service level agreements with possibly several providers state:

- *WWW meta-class:* At least 10% of the link bandwidth must be available for WWW traffic, divided into 6% for HTTP traffic and 4% for FTP traffic. If surplus bandwidth is available, WWW traffic will be allowed to occupy the whole link bandwidth. Green-only HTTP traffic should not experience longer delays than one second. In the FTP case, this bound increases to five seconds.
- *Transactions:* Transaction traffic should at least experience the service of a virtual leased line with 128 KBit/s bandwidth. It must not exceed 500 KBit/s.
- *Media meta-class:* This meta-class consists of voice signalling and video traffic classes. Video packets must not experience longer delays than $40\ ms$. Video must at least receive 128 KBit/s of the link bandwidth. For voice signalling, 64 KBit/s are reserved. Media-traffic is upper-bounded by 500 KBit/s.

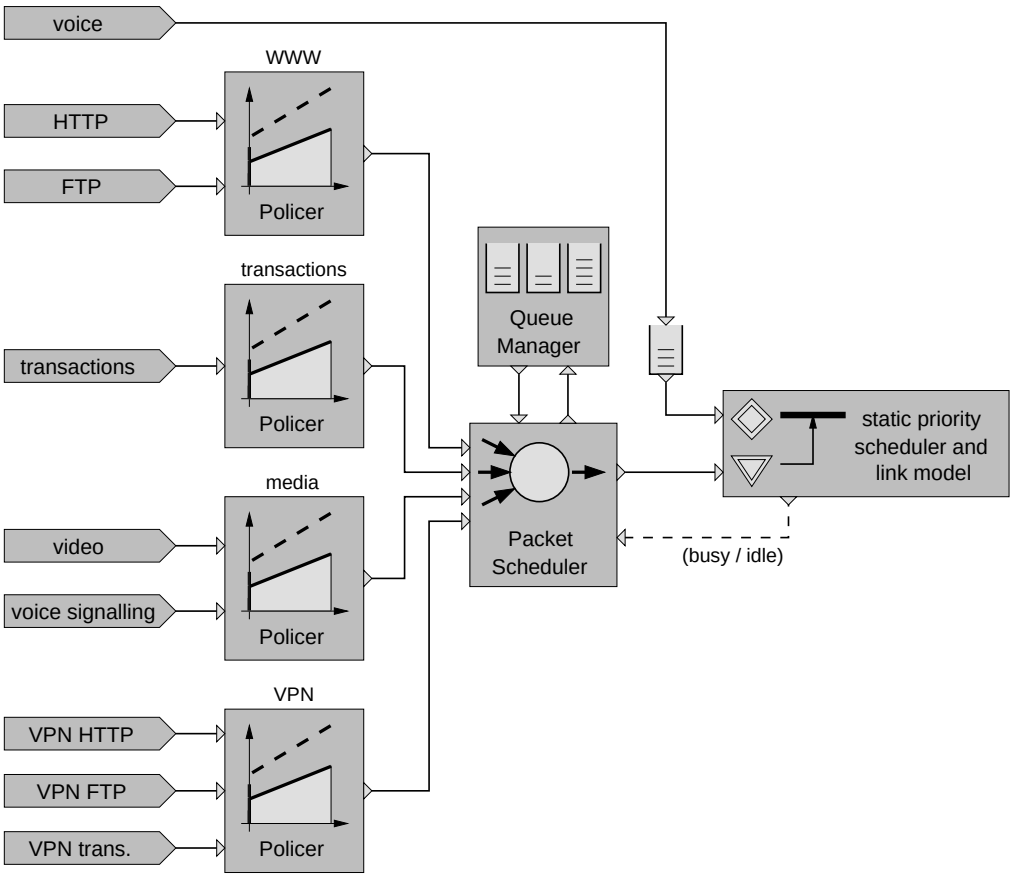

Fig. 25: Overview of the system for evaluation. The process network model shows the interconnection of policers, queue manager, and link scheduler.

- *VPN meta-class:* VPN traffic, consisting of FTP, HTTP, and transactions, must not surpass half of the link bandwidth. At least 10% of the link bandwidth must be available for VPN WWW traffic, divided into 6% for HTTP traffic and 4% for FTP traffic. VPN transaction traffic should experience the service of a virtual leased line with 128 KBit/s bandwidth.

Two types of service reservations can be distinguished. The first type specifies a minimal rate $r_{g,i}$ together with a maximum delay bound $d_{g,i}$. With eq. (3.1), the corresponding burstiness parameter $\sigma_{g,i}$ to configure the policer $(\sigma_{g,i}, \rho_{g,i})$ can be determined by

$$\sigma_{g,i} = (d_{g,i} - \frac{L}{R}) \cdot r_{g,i}$$

where L is the maximum packet length and R is the link rate. The rate $\rho_{g,i}$ can simply be set to the rate $r_{g,i}$. The second type of service reservation only mentions a minimum bandwidth $r_{g,i}$. In order to derive the burstiness parameter $\sigma_{g,i}$ for the policer, some properties of the incoming traffic must be known. For instance, in the voice signalling case, it is known that bursts may only consist

of a small number of packets. Then, a suitable $\sigma_{g,i}$ can be set and a delay bound can be derived by eq. (3.1). Again, the policer's rate $\rho_{g,i}$ is set to $r_{g,i}$.

3.3.2.2 Required context RAM space

Before the results from simulation runs will be described in detail, the memory space required to store parameters and variables per component is roughly estimated based on the description of the algorithms in the preceding section. The following data type precisions and parameters are assumed:

Maximum number of QoS classes	128
Maximum number of meta-classes	32
Maximum number of packets to be buffered by the QM	10000
Payload pointer size	32 Bit
Pointer used only within context RAM	16 Bit
Integer field precision	32 Bit
Floating-point precision	64 Bit
Packet length descriptor	11 Bit
Color mark	2 Bit
Flow identifier	7 Bit

The following RAM space bounds can then be derived, see Tab. 6.

Tab. 6: Bounds for the required space of the context RAM per component.

Policer	
Nested token buckets	5 KByte
Link Schedulers	
Deficit Round-Robin (DRR)	2 KByte
Self-Clocked Fair Queuing (SCFQ)	5 KByte
Minimum Delay SCFQ (MD-SCFQ)	5 KByte
Starting Potential-based Fair Queueing (SPFQ)	7 KByte
Queue managers (QMs)	
FIFO and yellow queues (needed by all QMs)	166.5 KByte
Free lists for yellow and FIFO queues (needed by all QMs)	39.5 KByte
Free list for payload RAM (needed by all QMs)	156.5 KByte
Central Yellow Queue (CYQ) QM	2 KByte
Enhanced Central Yellow Queue (CYQ-enh.) QM	2.5 KByte
CYQ with Random Early Detection (CYQ-RED) QM	2 KByte
Fair per-flow yellow queue QM	4 KByte

Notes:

- *Link scheduler:* In addition to the data structures mentioned in Subsection 3.2.1.2, a scheduler uses a set of backlog counters which keep track of

the current number of packets in the QoS classes. With the help of these counters, the scheduler can quickly determine the backlog state of a flow without the necessity to ask the queue manager.

A priority queue which is implemented by a heap-organized array uses entries that consist of a payload pointer, a scheduling tag, and a flow identifier. The size of a priority queue is below 2 KByte and included in the numbers of Tab. 6. Recall that SPFQ requires two priority queues.

An entry in the active list of a DRR scheduler consists of a flow identifier and a pointer to the next element.

- *Queue manager:* An entry in the priority queue of a fair queue manager consists of an overload value and a flow identifier.

  Three different free lists are required. A list is employed to stack addresses of free entries in the context memory that are used to store packet header information in a flow's queue. A list of free entries is needed which are used to enqueue packet header information in a yellow queue. A third list of free addresses of payload RAM entries is needed to store the actual contents of a packet. The latter list manages 68 Byte segments of the payload RAM. 64 Byte of a segment are usable for storing the contents of a packet and 4 Byte are reserved for an optional pointer to the next segment if the packet contents are divided among several segments. The number of addresses held by this free list may be up to four times the maximum number of packets – 40000 in our case. Recall that the payload memory is a RAM separated from the context RAM.

  An entry in a double-linked yellow queue consists of three pointers: two pointers to maintain the data structure and another pointer to the corresponding entry in the FIFO queue. As a result, an entry in the yellow queue needs 6 Byte.

  An entry in a FIFO queue of a flow consists of a pointer to the payload, a pointer to the next element in the queue, a pointer to a potential entry in the yellow queue, a packet length field, a color mark, and a flow identifier which sums to 11 Byte for a FIFO queue entry representing a packet. In the worst-case, a packet occupies an entry in the FIFO queue of its flow and an additional entry in the yellow queue.

- *Code memory:* The algorithm models cannot provide any precise estimates of the required code memory. We however want to point out the Java byte-code size of the performance models which may give us a hint of the order of magnitude of the needed code memory. The policer's code requires 7.4 KByte, the queue manager's code up to 14.4 KByte, and the link scheduler's code up to 13.8 KByte respectively. These code segments contain some overhead to interface with the simulation environment but do not include code for priority queues and dynamic memory allocation. These two subtasks are rather simple compared with, for instance, link scheduling. We thus believe that an additional 64 KByte of code memory should be more than enough to implement policing, queue management, and link scheduling in software.

The RAM space required to store parameters and variables is rather small for all components and scales linearly with the number of QoS classes to support. These RAM areas can probably be implemented with on-chip RAM. The data structures however that facilitate dynamic memory allocation and release scale linearly with the number of packets to be buffered. Their memory consumption can be rather high. The same statement is true for the payload memory. Since 40000 payload segments can be stored in our configuration, the payload needs 2.6 MByte of memory which clearly favors an off-chip DRAM implementation. The payload free list should also be mapped onto the payload memory in this case. Since the payload memory must only support a throughput of approx. twice the access link bandwidth to store and read packets at link speed, the payload memory will rather be bounded by capacity than by bandwidth utilization.

3.3.2.3 Simulation results for a 10 MBit access link

This link bandwidth models future access links, e.g. provided by wireless networks. The simulation settings assuming a 10 MBit/s access link are listed in Tab. 7. The policer settings as well as the worst-case delay bounds are derived from the requirements of the SLAs. Individual buffer requirements for the QoS classes can be determined by $\sigma_{g,i} + \frac{L}{R} \cdot \rho_{g,i}$ according to eq. (3.2) where L is the maximum packet length and R is the link rate respectively. For our buffer reservations, the second term is rounded up to L and another L is added to account for possible blocking due to high-priority voice traffic. Thus, the buffer requirements are derived by adding two maximum packet lengths to the burstiness $\sigma_{g,i}$ at the policer. The rate reservations at the scheduler are augmented by 1 % compared with the policer settings to allow queues to shorten their lengths in times of heavy utilization by green traffic. The quantum values for DRR are derived from the maximum packet length of 1.5 KByte. The shared memory of 670 KByte is dimensioned to just hold the sum of the green profiles or the maximum yellow profile. In this way, the queue manager takes a big part of the responsibility for the recovery from congestion.

Traffic traces (see Tab. 4) stimulate the system for a simulated time of 30 seconds. The system swings out afterwards for another 1.5 seconds to empty buffers. Load values for architecture blocks are calculated every 1 ms. Different simulation runs are performed in which additional voice traffic – an aggregation of eight voice sources – is fed in and/or the amount of video traffic varies.

Observation: With the given simulation settings, none of the combinations of policing, queuing, and scheduling shows congestion. Thus, only the policer has to drop packets, but not the queue manager. Therefore, the delay experienced by the flows is only influenced by the interplay of policer and link scheduler.
Delay properties: In Fig. 26, the maximum delays experienced by packets of different QoS classes are displayed. Four system configurations are considered which are defined by the choice of a link scheduler. One can state:

- All delays are far below the worst-case calculation.
- WFQ-based systems – MD-SCFQ, SCFQ, and SPFQ – punish the video traffic

Tab. 7: Simulation settings for a 10 MBit/s link. Policer buckets are described by the burstiness σ and long-term bounding rate ρ. The WFQ weights Φ_i are scaled to the interval $[0 \ldots 1)$. The DRR quantum values q_i denote the portions of the overall Round-Robin frame. The worst-case delay for a WFQ-based system is derived by eq. (3.1) assuming no high-priority voice traffic.

QoS classes		*Policer buckets* (σ, ρ) $[Byte, Bit/s]$	*QM allocation* $[Byte]$	*Scheduler* *WFQ* Φ_i	*Scheduler* *DRR* $q_i\,[Byte]$	*WFQ wc-delay* $[ms]$
WWW	*meta-class*	$(610 \cdot 10^3\ ,\ 10 \cdot 10^6)$	(670K)			
	HTTP	$(\ 74908\ ,\ 600 \cdot 10^3)$	77980	0.0606	14400	991
	FTP	$(\ 249939\ ,\ 400 \cdot 10^3)$	253011	0.0404	9600	4951
Trans.	*meta-class*	$(\ 6000\ ,\ 5 \cdot 10^5)$	(670K)			
	Transactions	$(\ 1375\ ,\ 128 \cdot 10^3)$	4447	0.012928	3072	87
Media	*meta-class*	$(\ 10000\ ,\ 5 \cdot 10^5)$	(670K)			
	Voice signal	$(\ 3200\ ,\ 64 \cdot 10^3)$	6272	$6.464 \cdot 10^{-3}$	1536	398
	Video	$(\ 621\ ,\ 128 \cdot 10^3)$	3693	0.012928	3072	40
VPN	*meta-class*	$(610 \cdot 10^3\ ,\ 5 \cdot 10^6)$	(670K)			
	VPN HTTP	$(\ 74908\ ,\ 600 \cdot 10^3)$	77980	0.0606	14400	991
	VPN FTP	$(\ 249939\ ,\ 400 \cdot 10^3)$	253011	0.0404	9600	4951
	VPN trans.	$(\ 1375\ ,\ 128 \cdot 10^3)$	4447	0.012928	3072	87

class with longer delays when the video rate is increased from 42 KBit/s to 298 KBit/s. In the former case, the video traffic fits well in its green profile while in the latter case the profile is exceeded by more than a factor of two. The delay properties of the other flows are not affected by the variation of video traffic. These flows are well isolated from the behavior of the video class.

- Opposed to WFQ-based systems, the DRR system shows a completely different behavior. Not only are the maximum delays of small bandwidth flows clearly above the delays experienced in WFQ-based systems, but also the delay does not increase for the video class with traffic exceeding the green profile. DRR thus rewards the greedy behavior of the video class.

- The high-priority voice traffic deranges all traffic classes. The overhead can be quite high for low bandwidth - low latency classes such as the transactions class. The maximum delay increases by up to 50% whereas the overhead in terms of delay varies from 7% to 18% for the high bandwidth FTP and HTTP classes.

  In order to compare the delay properties of all systems, Fig. 27 shows the maximum delays scaled to the delays in the MD-SCFQ-based system. MD-SCFQ is chosen as reference because it belongs to the class of rate proportional servers [152] and therefore shows the same worst-case delay properties as the ideal WFQ. From Fig. 27 one can see that:

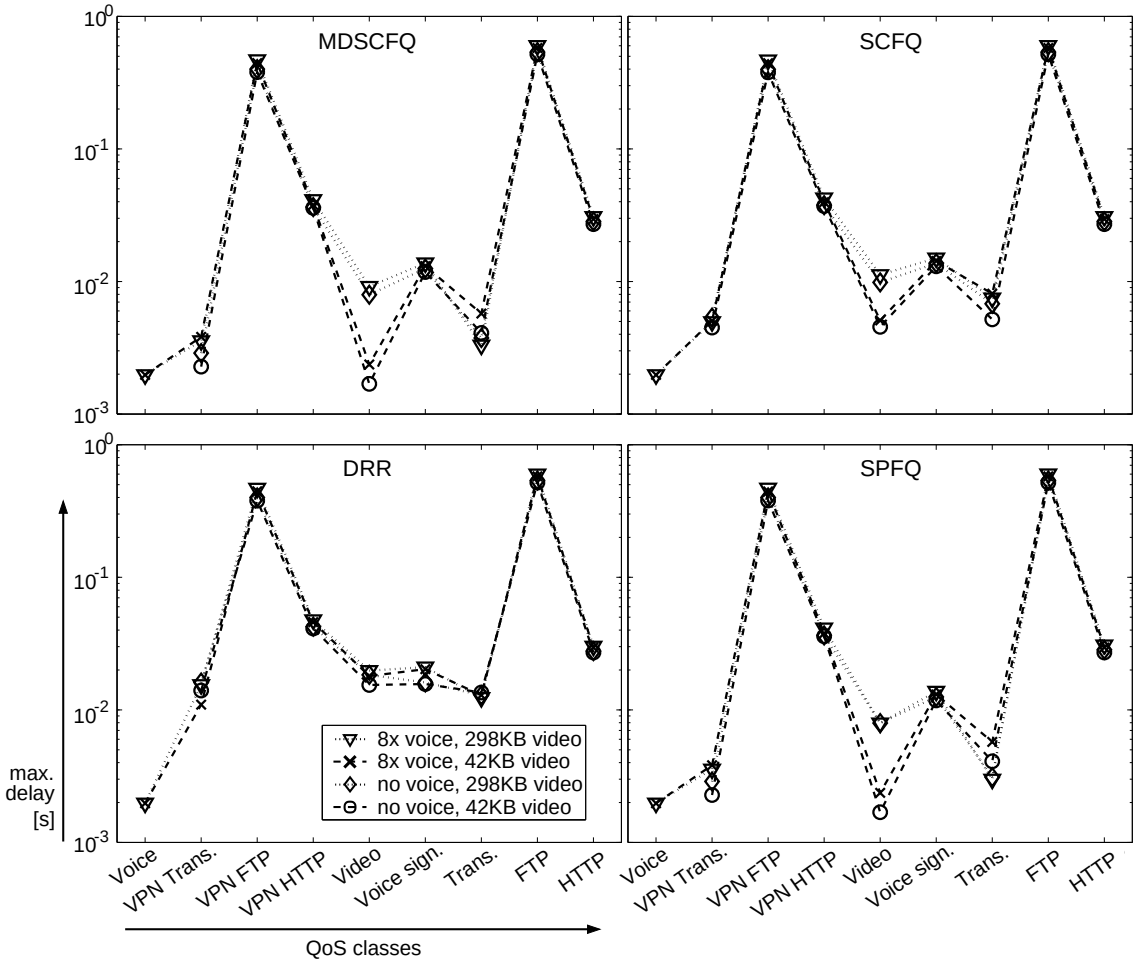

Fig. 26: Maximum delay experienced by the flows using a 10 MBit/s link without the occurrence of congestion. Since the timing behavior of the system does not depend on the choice of the queue manager, only four configurations are shown which differ in the employed link scheduler. The amount of voice and video traffic is varied.

- SCFQ and DRR handicap small bandwidth - low delay traffic classes. However, the effect is significantly less for SCFQ than for DRR.

- MD-SCFQ and SPFQ perform quite similarly in terms of maximum delay.

Resource utilization: Tab. 8 lists the maximum load experienced by architecture blocks. Since the system is quite underutilized, only the results for the slowest architecture blocks are given. That means, the maximum load for the components – policer, queue manager, and link scheduler – is listed when each component is mapped onto its own ARM CPU core with an assigned SDRAM as well as when the components share a single ARM CPU core and a single SDRAM. There is always a separate SDRAM for the payload. Moreover, the maximum load caused by priority queues in the fair queue manager and in all WFQ-based schedulers is shown. The following can be observed:

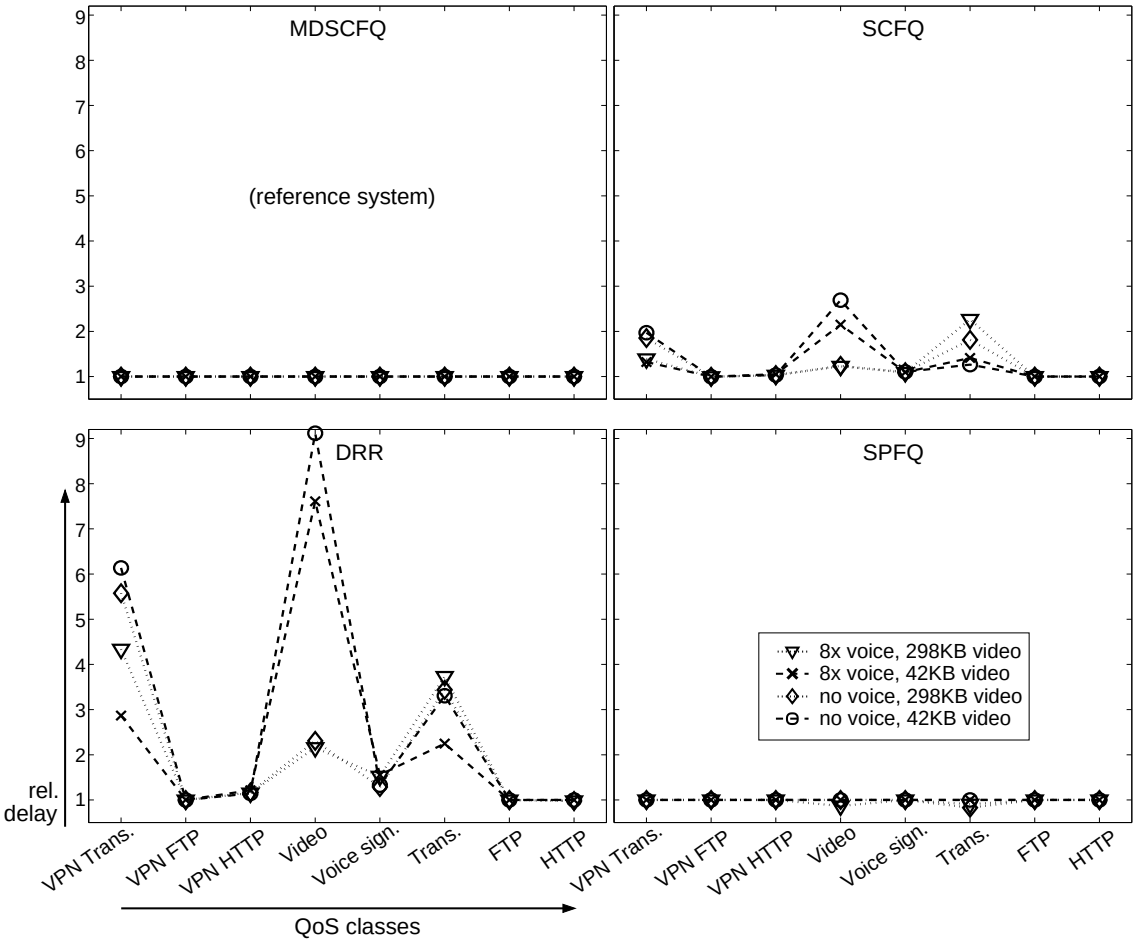

Fig. 27: Maximum delay experienced by the flows at an uncongested 10 MBit/s link in comparison with the MD-SCFQ-based system.

Tab. 8: Maximum load of architecture blocks for the 10 MBit/s link. Only the load of the slowest architecture models are shown, i.e., for ARM CPU cores and SDRAMs.

	Scheduler				*Queue Manager*				*Policer*	*Prio-Queue*	*Overall*
Load [%]	*MD-SCFQ*	*SCFQ*	*DRR*	*SPFQ*	*CYQ*	*CYQ-enh.*	*CYQ-RED*	*YQ-Fair*			
CPU	6.1	2.5	0.2	3.1	1.2	1.2	2.1	5.0	2.8	0.7	8.7
RAM	1.6	1.3	0.2	1.5	6.1	6.1	7.0	6.3	1.1	0.6	7.9
Payload											3.4

- The scheduler generating the highest CPU load is MD-SCFQ. Although SPFQ requires a second priority queue, MD-SCFQ's calculation of the system potential seems to be a greater burden.
- DRR causes considerably lower load than WFQ-based schedulers.
- Maintaining per-flow state for yellow traffic in the queue manager affects the

CPU load by more than a factor of two. The RAM overhead can be neglected.

- The CPU load is quite balanced between the scheduler, the queue manager, and the policer. The RAM load however is dominated by the queue manager.

- The sum of the maximum per-component load values is significantly greater than the maximum load of a single, shared CPU and a single RAM. That means, the load peaks of the components do not appear at the same time and one can benefit from sharing computing resources.

Discussion: *Delay properties:* DRR cannot support as tight delay bounds as the WFQ-based schedulers can, since the minimum slot length within a Round-Robin frame should be at least one maximum packet length [142]. In this way, frames become very large for small bandwidth - small delay flows. For instance, in our case (Tab. 7) a transmission of a frame may take about 50 ms. This however is already of the same order of magnitude than WFQ's delay bounds for the low delay classes for transactions and video traffic.

The aggregated voice traffic consists of eight constant bit rate voice sources. On the average, it occupies 8.2% of the link bandwidth. A burst of eight voice packets takes about 1 ms. These bursts appear every 10 ms. Therefore, the service of a backlogged queue with assigned worst-case delay above 9 ms may be interrupted for several times by voice traffic until a packet is transmitted.

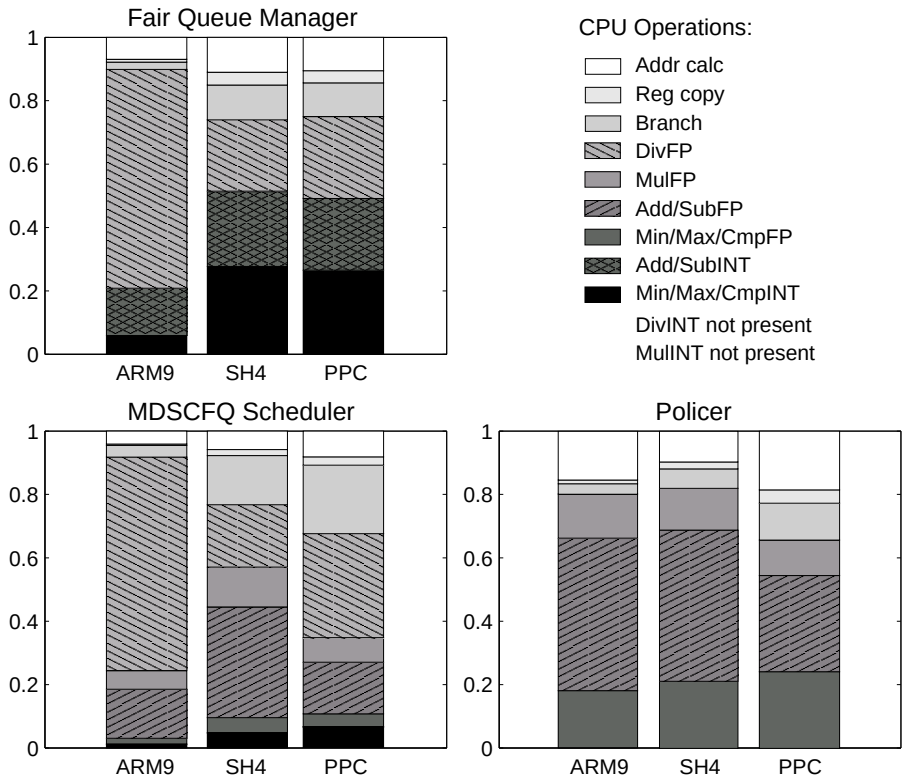

Fig. 28: Normalized overall execution time for selected components divided among operation types for different CPU models.

Resource utilization: The load generated by a component heavily depends on the share of floating-point operations. Fig. 28 illustrates an example. The overall run-time profile generated by a WFQ-based scheduler, the fair queue manager, and the policer are displayed. The overall execution time is divided among the monitored CPU operations and shown for all implemented CPU models. If MD-SCFQ or a fair queue manager are mapped onto an ARM core, the load will be determined by more than 70% by floating-point division operations. This share can be drastically reduced, for instance, by employing a CPU with a dedicated floating point unit such as an SH4 or a PowerPC. The CPU load peaks can thus be reduced by more than a factor of three (not displayed). The policer's load is strongly influenced by the performance of floating-point additions and min/max operations. The impact of these operations cannot largely be reduced by changing the CPU. The load peaks rather decrease due to higher clock rates.

Marking statistics of different traffic traces after policing are displayed in Fig. 29. The transactions traces, the voice signalling traces, the 42 KBit/s video trace, and the HTTP traces can be considered to comply with their green profiles. The FTP traces however represent greedy, unresponsive flows since more than 60% of the traffic is already dropped at the policer and more than half of the remaining traffic only fits into the meta-class profile. By increasing the video traffic from 42 KBit/s to 298 KBit/s, the amount of green-marked video traffic is more than halved. However, the traffic still fits into the meta-class profile and no packet is dropped at the policer. Together with the policer statistics, we have another indication that the system is actually underutilized. Although the FTP traces and the 298 KBit/s video trace do not keep their green profiles, the system still meets the assigned worst-case delays for these flows.

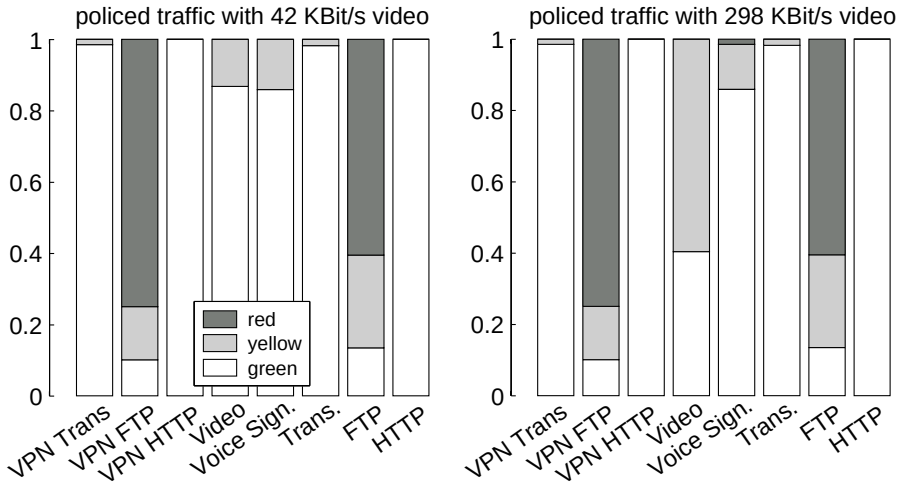

Fig. 29: Traffic statistics after policing. The relative amount of green-, yellow-, and red-marked traffic is shown for each QoS class. Video traffic is varied from 42 KBit/s to 298 KBit/s.

Conclusion for the non-congested 10 MBit link:

- Although congestion does not occur, the highest load for a RAM component is already generated by the queue manager for maintaining per-flow queues.

- The CPU load is rather balanced among scheduler, policer, and queue manager. Thus, one cannot achieve much speed-up by just optimizing a single component. Since all components show a high dependence on floating-point operations, one could either spend a dedicated floating-point processing unit or further investigate the replacement of floating-point operations by integer or fixed-point ones. The latter case would of course react upon the timing behavior of the algorithms.

- Although the number of classes is small, DRR shows considerably worse delay properties than WFQ-based schedulers. DRR however generates less than one fifth the load for RAM and CPU than WFQ-based schedulers.

- The best delay properties are supplied by MD-SCFQ and SPFQ. SPFQ causes lighter load than MD-SCFQ although SPFQ utilizes two priority queues.

3.3.2.4 Simulation results for a 2 MBit access link

Further simulation results for a non-congested system are presented with a reduced link rate of 2 MBit/s modeling forthcoming high-bandwidth cable modem and DSL solutions. The discussion focuses on the load of CPU and RAM components to derive how strongly the load scales with the supported link rate. Since the same SLAs, traffic traces, and minimum and maximum packet lengths are employed as in the preceding subsection, the settings for the policer, queue manager, and packet scheduler must be adjusted accordingly. The settings for a 2 MBit link are given in Tab. 9. Since some of the traffic classes specify a fixed rate in the SLA such as the video and the transaction classes, their share of the link bandwidth now increases.

Observation: *Delay properties:*

- None of the simulated configurations shows congestion as long as no voice traffic is fed in. Moreover, all worst-case delays are met again, although the FTP, HTTP, and video classes have a high amount of yellow-marked traffic.

- With additional voice traffic, the queue manager must drop up to 18% of the yellow-marked traffic from the FTP classes. The video class no longer keeps its worst-case delay when video traffic with an average rate of 298 KBit/s is fed in so that the green traffic profile is not matched any more.

- The maximum delay penalty of small bandwidth - small delay flows caused by the DRR packet scheduler reduces by a factor of two to three compared with the 10 MBit link settings. The maximum delays experienced in a DRR system are however still by a factor two to four worse than in the WFQ-based systems.

Resource utilization: The maximum load values experienced by the architecture blocks in absence of voice traffic are listed in Tab. 10. Since the system is rather underutilized, only the results for the slowest architecture blocks – ARM CPU cores and SDRAMs – are given. Compared with the results of the 10 MBit link in Tab. 8, one recognizes that:

Tab. 9: Simulation settings for a 2 MBit/s link. Policer buckets are described by the burstiness σ and long-term bounding rate ρ. The WFQ weights Φ_i are scaled to the interval $[0 \ldots 1)$. The DRR quantum values q_i denote the portions of the overall Round-Robin frame. The worst-case delay for a WFQ-based system is derived by eq. (3.1) assuming no high-priority voice traffic.

QoS classes		*Policer buckets* (σ, ρ) $[Byte, Bit/s]$	*QM allocation* $[Byte]$	*Scheduler* *WFQ* Φ_i	*Scheduler* *DRR* $q_i\ [Byte]$	*WFQ wc-delay* $[ms]$
WWW	*meta-class*	$(150 \cdot 10^3\ ,\ 2 \cdot 10^6)$	(160K)			
	HTTP	$(\ 14908\ ,\ 120 \cdot 10^3)$	17980	0.0606	2880	990
	FTP	$(\ 49939\ ,\ 80 \cdot 10^3)$	53011	0.0404	1920	4951
Trans.	*meta-class*	$(\ 6000\ ,\ 5 \cdot 10^5)$	(160K)			
	Transactions	$(\ 1375\ ,\ 128 \cdot 10^3)$	4447	0.06464	3072	92
Media	*meta-class*	$(\ 10000\ ,\ 5 \cdot 10^5)$	(160K)			
	Voice signal	$(\ 3200\ ,\ 64 \cdot 10^3)$	6272	0.03232	1536	403
	Video	$(\ 542\ ,\ 128 \cdot 10^3)$	3614	0.06464	3072	40
VPN	*meta-class*	$(150 \cdot 10^3\ ,\ 10^6)$	(160K)			
	VPN HTTP	$(\ 14908\ ,\ 120 \cdot 10^3)$	17980	0.0606	2880	990
	VPN FTP	$(\ 49939\ ,\ 80 \cdot 10^3)$	53011	0.0404	1920	4951
	VPN trans.	$(\ 1375\ ,\ 128 \cdot 10^3)$	4447	0.06464	3072	92

Tab. 10: Maximum load of architecture blocks for the 2 MBit/s link. Only the load for the slowest architecture models are shown, i.e., for ARM CPU cores and SDRAMs.

	Scheduler				*Queue Manager*				*Policer*	*Prio-Queue*	*Overall*
Load [%]	*MD-SCFQ*	*SCFQ*	*DRR*	*SPFQ*	*CYQ*	*CYQ-enh.*	*CYQ-RED*	*YQ-Fair*			
CPU	3.8	1.9	0.2	2.6	1.0	1.0	1.9	4.6	2.8	0.8	7.2
RAM	1.5	1.3	0.1	1.4	5.1	5.1	5.6	6.8	1.1	0.8	7.8
Payload											2.7

- The CPU load for the scheduling components roughly decreases by one third whereas the RAM load reduces by less than 10%.
- The load caused by the queue manager reduces by 10% to 20%.
- The CPU load remains quite balanced among the different components and the RAM load is especially caused by the queue manager.

  Discussion: *Delay properties:* The reduction of DRR's delay penalty is not caused by DRR itself. Since the reservations of the flows which are penalized most by DRR – video and transactions – are not changed compared with the 10 MBit link settings, the negative impact of the Round-Robin frame length is reduced since these flows now reserve a greater share of the link bandwidth.

The additional aggregated voice traffic is a great burden for the 2 MBit link. Eight constant bit rate voice sources occupy more than 40% of the link bandwidth which is almost as much as the reservation for all green profiles that take up 43% of the link bandwidth. The transmission of a burst of eight voice packets takes 4 ms. These bursts occur every 10 ms. Consequently, since the voice traffic is not regulated by the packet scheduler, the maximum experienced delay at least doubles for all flows. Even flows are affected that fit well within their green profile due to the fixed higher priority of voice traffic.

The transmission of a packet with maximum length 1.5 KByte already takes 6 ms. This is already of the same order of magnitude as the delay experienced by low delay flow classes. Since a transmission cannot be preempted, the maximum delay that, for instance, a transaction class suffers is virtually more than doubled compared with the 10 MBit link.

Resource utilization: The maximum CPU and RAM load values only moderately decrease with the smaller link rate because the load peaks appear after an idle period when the token buckets of the policer are filled up. Then, traffic may pass with its peak rate generating relatively high load for the queue manager

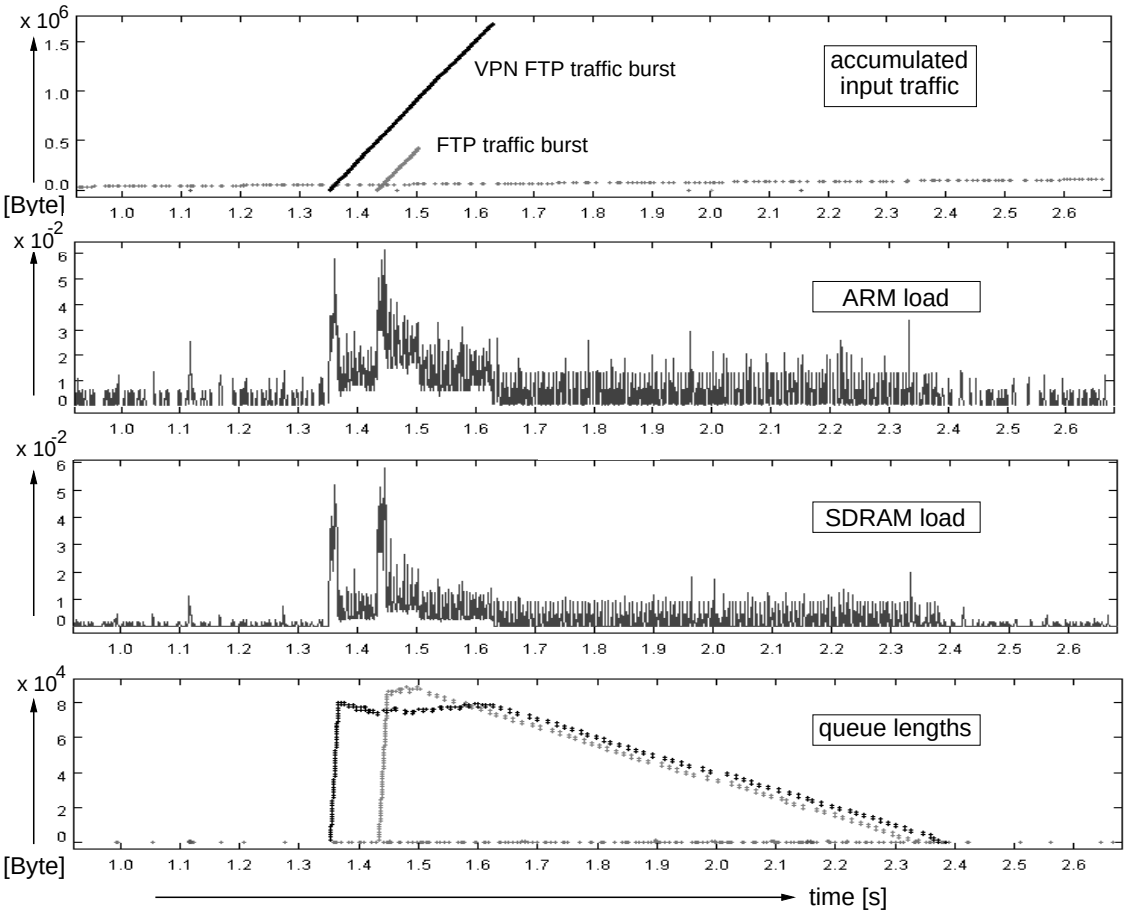

Fig. 30: Excerpt from an execution trace: Load peaks are shown that appear at the beginning of a busy period when traffic may fill up the queue memory at its peak rate.

and the scheduler. An example from an execution trace is shown in Fig. 30. Two traffic bursts appear at time 1.35 s and 1.43 s respectively. The queues are empty beforehand. About one tenth of a second after the beginning of a burst, the policer begins to limit the incoming traffic to the reserved rate. This is why the queue lengths then stagnate for the rest of the burst period. After the end of a burst, the backlog in the corresponding queue can gradually be served. The load peaks appear at the beginning of the busy period when a burst fills the memory up at its peak rate until the policer finally limits the incoming traffic.

Conclusion for the non-congested 2 MBit link:

- The supported link rate does not heavily affect the maximum load of the resources since the load peaks are experienced at the input side of the system. To decrease the influence of the peak rate of incoming traffic, the policer could use a more complicated traffic profile – for instance a TSpec – that not only bounds the burstiness but also the peak rate of a flow. This may decrease the pressure on the queue manager but would in turn rise the load of the policer.

- Eight uncompressed voice sources with fixed highest priority already are a great burden for the system. Guarantees and stability of the system are jeopardized.

- A non-preemptive transmission of a packet with maximum length may considerably impair the delay properties of a flow. This overhead can be reduced by fragmentation at the expense of additional packet headers.

3.3.2.5 Simulation results for a 10 MBit link with flooding

With the following simulation runs worst-case configurations are investigated. The system is stimulated by a flood of very small packets which results in congestion. It is assumed that the system has three incoming links that correspond to a virtual private network, video sources, and other network traffic. The settings for policer, queue manager, and link scheduler are retained according to Tab. 7. Traffic traces are assigned to the QoS classes in the following way:

QoS class	*Trace type* (see Tab. 4)
HTTP	flooding trace
FTP	transactions-like
Transactions	transactions-like
Voice signal.	Voice signalling
Video	flooding trace
VPN HTTP	flooding trace
VPN FTP	transactions-like
VPN trans.	transactions-like

Additionally, aggregated voice traffic is fed in. Since the flooding traffic takes up all incoming links, the remaining classes are only stimulated by small bandwidth traces. This traffic therefore is compliant with the green profiles and

green-marked completely. The flooding traffic can only be bounded to some extent by the meta-classes and thus consists of a high amount of yellow traffic.
Observation: *Delay properties:* In Fig. 31, the maximum delays experienced by the flows for all combinations of link schedulers and queue managers are shown. The values are scaled to the delays experienced in a system with a MD-SCFQ scheduler and a fair queue manager. The peak rate of the stimulating flooding traces is varied from 10 MBit/s over 32 MBit/s to 50 MBit/s. In a mixed setting, the HTTP class is fed by a 50 MBit/s trace, the video class by a 32 MBit/s trace, and the VPN HTTP class by a 10 MBit/s trace respectively. The flooding trace characteristics have been introduced in Tab. 4. The following description is restricted to relative delays since the absolute delay values do not reveal any new insights which are specific for the congested system. All deadlines are met for traffic that completely complies with green profiles. For instance, the maximum delay experienced by the transactions class is only about 10 ms in a WFQ-based system. This value increases by the factor 2.5 in a DRR system. The relative delay behavior displayed in Fig. 31 shows the following:

- SCFQ together with a fair queue manager performs as well as the other WFQ-based systems during times of congestion. This was not the case in the uncongested scenario where delays were worse by up to a factor of three.

- DRR again extremely punishes some of the classes. Moreover, the combination with a fair queue manager cannot improve this property.

- MD-SCFQ and SPFQ again show virtually equal delay properties.

- The queue managers that do not distinguish per-flow state for yellow traffic – CYQ, CYQ-enh., and CYQ-RED – show a uniform behavior: one of the queues that are fed by a flooding source is heavily punished in terms of delay in favor of another queue that also holds traffic from a flooding source.

  Resource utilization: A detailed listing of the maximum load values for different architecture blocks derived from simulations with 50 MBit/s flooding input traces is shown in Tab. 11 and Tab. 12. The former table shows the maximum load values if the components of the system – policer, queue manager, and link scheduler – are mapped onto individual CPU and RAM blocks as well as if all components are mapped onto the same CPU and RAM resources. The latter table illustrates the maximum load generated by special blocks which implement dynamic memory allocation and priority queuing as described in Subsection 3.2.2. Their load is included in the respective component load in Tab. 11. For the priority queue, we consider two subcases. First, the packet scheduler SPFQ needs two priority queues opposed to SCFQ and MD-SCFQ that only require a single priority queue. Secondly, the combination of SPFQ with a fair queue manager needs a third priority queue. Moreover, the RAM load generated for payload storage is displayed. The packet payload is stored separately from parameters and variables of the system. The following can be remarked:

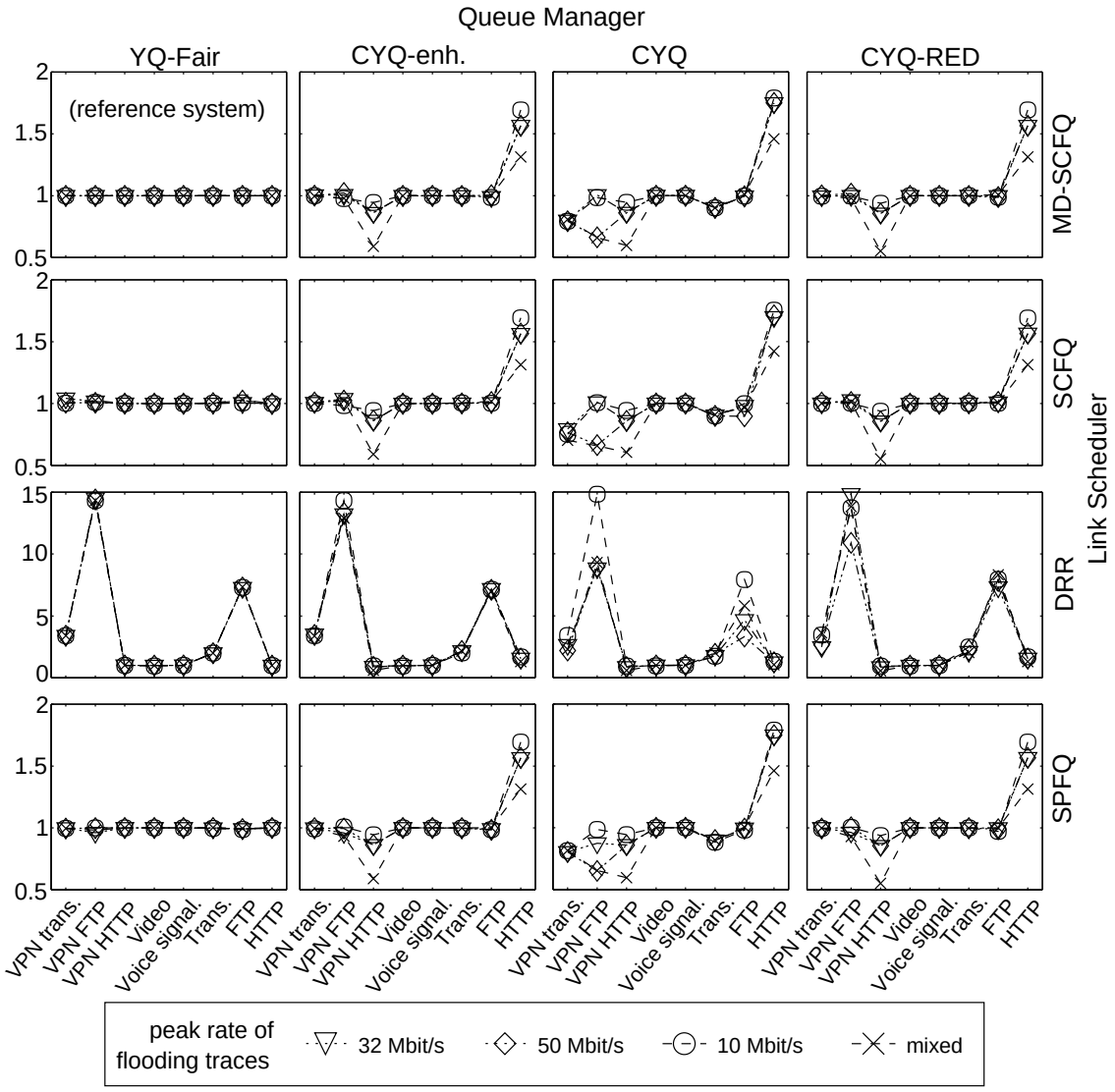

Fig. 31: Maximum delay experienced by the flows at a 10 MBit/s link scaled to the MD-SCFQ system with a fair queue manager. Horizontally aligned diagrams belong to the same link scheduler whereas vertically aligned graphs belong to the same queue manager. The peak rate of the flooding traces (Tab. 4) is varied from 10 MBit/s over 32 MBit/s to 50 MBit/s. In the *mixed* setting, the HTTP class is fed by a 50 MBit/s trace, the video class by a 32 MBit/s trace, and the VPN HTTP class by a 10 MBit/s trace respectively.

- The difference in the load caused by DRR and WFQ-based schedulers decreases. However, DRR's load still is roughly one third of SCFQ's load.
- The system may be overloaded not only if all components are mapped onto shared resources but also if the components are mapped to separate CPUs and RAMs. Due to the load generated by the policer, the system cannot be implemented by ARM cores only.
- Maintaining per-flow states for yellow traffic in the queue manager during times

Tab. 11: Maximum load of architecture blocks for the 10 MBit/s link. The system is flooded by maximum small packets at 50 MBit/s peak rate.

	Scheduler				*Queue Manager*				*Policer*	*Overall*
Load [%]	*MD-SCFQ*	*SCFQ*	*DRR*	*SPFQ*	*CYQ*	*CYQ-enh.*	*CYQ-RED*	*YQ-Fair*		
CPU blocks										
ARM	49.6	20.1	7.4	39.1	19.5	20.6	99.0	153.8	100.8	302.5
SH4	11.9	5.1	4.0	11.5	6.4	6.9	18.8	20.3	32.4	63.9
PPC	3.3	2.0	1.6	3.2	2.6	2.8	6.1	7.8	6.8	17.6
RAM blocks										
SDRAM	16.8	4.3	0.8	17.1	58.0	58.0	89.1	93.5	37.3	146.9
SRAM	6.8	1.8	0.3	7.0	23.0	23.0	34.5	37.2	14.9	58.6

Tab. 12: Maximum load of architecture blocks for the 10 MBit/s link. The system is flooded by small packets at 50 MBit/s peak rate. The load generated by special hardware blocks is listed. Their load values are included in the load of the respective component in Tab. 11 (besides payload storage).

Load [%]	*Payload*	*Priority queue*		*Dynamic memory allocation*
		SPFQ + YQ-Fair	*SPFQ only*	
CPU blocks				
ARM	–	30.2	5.6	3.3
SH4	–	10.5	2.1	1.0
PPC	–	3.4	0.7	0.4
RAM blocks				
SDRAM	5.5	33.0	5.3	23.4
SRAM	3.1	13.3	2.1	8.8

of congestion enormously increases the maximum CPU load by a factor of seven and the RAM load by about one half.

- The queue manager CYQ-RED employs RED on a single yellow queue without maintaining per-flow state. Nevertheless, CYQ-RED is almost as complex as the fair queue manager.
- The CPU load is especially determined by the policer and the choice of the queue manager.
- The RAM load is defined for the most part by the queue manager.
- By using a CPU with a dedicated floating-point unit, the system can be implemented by a single, shared CPU chip.
- One cannot gain much in efficiency in times of congestion by sharing resources.

The maximum load derived for a single, shared CPU and a single, shared RAM is almost as bad as the sum of the respective load values of distributed resources.

- A 10 MBit/s link does not burden an SDRAM-based payload RAM very much.
- (Not displayed) The maximum load appears again at the beginning of a busy period when the policer allows traffic to pass with its peak rate.
- (Not displayed) When the peak rate of the flooding traces at the input is reduced from 50 MBit/s to 10 MBit/s, RAM and CPU load values of the overall system reduce to one third. In detail, the maximum load caused by the policer decreases by a factor of five, the QM's load by a factor of 3.5 to four, and the scheduler's load up to a factor of 1.6 respectively.

Discussion: *Delay properties:* Recall the latency dependencies in Tab. 1. SCFQ's worst-case delay bound not only depends on the number of flows sharing the link but also on the maximum packet length. Maximum small packets may not be the most challenging configuration for SCFQ in terms of timing. Some additional simulation runs have been performed in order to increase the impact of this term. The flooding traces with maximum small packets are replaced by flooding traces with maximum large packets maintaining peak rates. Looking at a system with a fair queue manager and an SCFQ scheduler, the delay properties become moderately worse. The almost ideal relative delay of one compared with the MD-SCFQ-based system increases to 1.5 for the voice signalling and the VPN transactions classes. However, this is still one order of magnitude better than the maximum factor 15 experienced in the DRR-based system. By using another queue manager than the fair queue manager the differences between the WFQ-based systems increasingly vanish.

In order to understand the timing of queue managers during congestion that do not maintain per-flow state for yellow traffic – i.e. CYQ, CYQ-enh., and CYQ-RED – one must additionally consider their dropping behavior. Fig. 32 shows drop statistics for combinations of the DRR and MD-SCFQ link schedulers with the four different queue managers. MD-SCFQ represents the WFQ-based systems that all basically show the same drop amounts. The share of dropped traffic from yellow-marked packets is shown in dependence on the used queue manager. The HTTP, VPN HTTP, and Video classes are fed by flooding traces. Since the meta-class to which HTTP belongs allows the largest amount of yellow traffic to pass the policer, the fair queue manager deletes most of the yellow traffic from this class. Note that none of the yellow packets from the video class must be dropped. This perfect isolation of flows however will vanish if a queue manager is employed that does not distinguish per-flow state for yellow traffic. One sees that the amount of traffic dropped from the VPN HTTP and video classes increases giving preference to the HTTP class. This effect will still be somewhat bounded if a WFQ-based link scheduler is employed. However, the flow isolation will virtually collapse if DRR is used. In the end, one can state that by using a somewhat unfair queue manager the most greedy

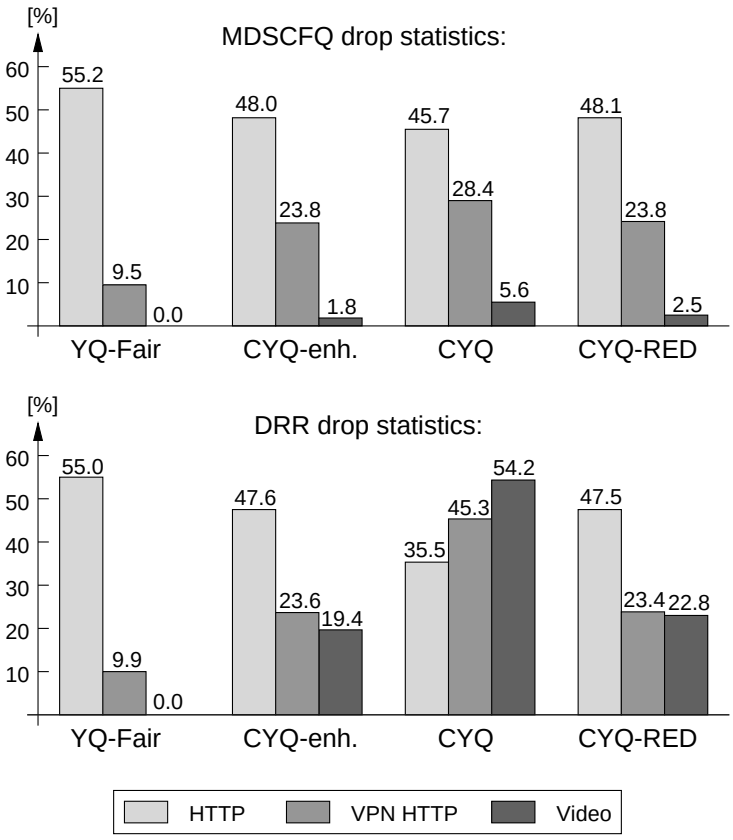

Fig. 32: Drop statistics for a congested 10 MBit/s link: the share of yellow traffic dropped by the queue managers is shown.

traffic class experiences longer maximum delays because less packets have been dropped from this class compared with a system that uses a fair queue manager. More packets must be dropped from other classes so that – as a side-effect – the remaining backlog of these classes may see smaller maximum delays.

Resource utilization: The fair queue manager employs a priority queue in which the flows are sorted according to their relative overload by yellow traffic. If a packet is dropped from a flow with the largest overload, its relative overload value must be dequeued from the priority queue, recalculated, and enqueued again. Thus, a single packet drop initiates several priority queue operations. This is why the impact of the priority queue used in the fair queue manager has a greater impact on the load than the priority queues in the link scheduler in Tab. 12. One further setting is investigated to explore whether the high impact of the fair queue manager on the RAM and CPU load due to packet drops can possibly be even higher. The system is again flooded by maximum small packets. After the shared memory has been filled up with small packets, the input traffic abruptly changes from maximum small to maximum large packets (1536 Byte). In this way, the enqueuing of a single maximum large packet may trigger the discard of more than 30 small packets. The resulting overall load of the system is displayed in Fig. 33. The length of the packets abruptly

changes after three seconds. However, the overall load decreases. Moreover, the per-component maximum load values do not increase either (not displayed). Working with maximum small packets therefore seems to be the right way to investigate the worst-case utilization of the system.

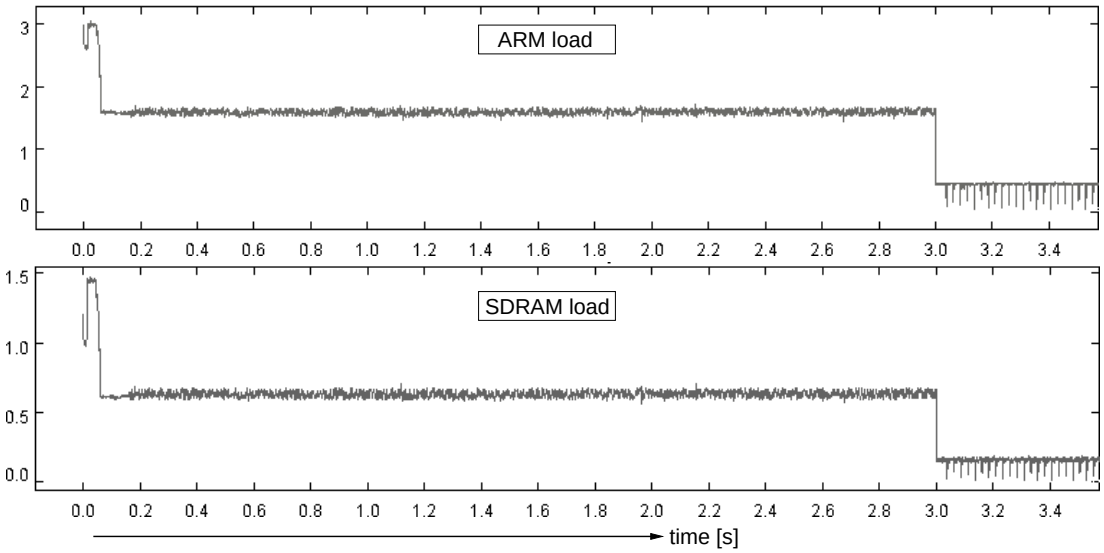

Fig. 33: Load for the flooded 10 MBit/s link. The MD-SCFQ scheduler and the fair queue manager are used. The system is stimulated by small packets at a peak rate of 50 MBit/s for three seconds. Then, the packet size changes to the maximum length while the peak rate is maintained.

The load caused by the fair queue manager could be reduced in many ways. The current calculation of the relative overload of a flow is based on floating-point computations. Moreover, one could apply an heuristic such as to only determine the flow with the highest overload once per packet arrival and not at every packet drop event. Besides exchanging floating-point computations by fixed-point or integer ones, we do not see much improvement potential for the RED-based queue manager. The RED implementation already includes most of the assumptions of the original paper ([54]) such as table-based random numbers and power-of-two approximations of exponential functions. Thus, the calculation of the average queue length required by RED generates considerable overhead close to the overhead experienced with the fair queue manager that maintains per-flow state for yellow traffic.

The high load values encountered for the policer in this study do not seem to be problematic for an actual implementation. On the one hand, token buckets can be easily implemented by programmable counters in hardware. On the other hand, the side-effects involved by replacing floating-point with another data type can best of all be estimated for the policer's computations.

Conclusion for the congested 10 MBit link:

- In times of congestion handling small packets, SCFQ performs just as well as

the other WFQ-based schedulers at half the load.

- DRR shows the advantage of low complexity. However, DRR's timing behavior is by far the worst one among all investigated scheduling algorithms. A sophisticated queue manager cannot ease DRR's bad timing.

- It is not worth implementing RED to avoid per-flow state distinction for yellow traffic. RED's complexity is already close to the complexity of a fair queue manager. However, by using the proposed fair queue manager, a better separation of tasks is achieved since packets are only dropped on arrival at the policer and not on arrival at the queue manager. The queue manager only pushes already stored packets out when congestion appears. In this way, we avoid to immediately drop packets which have just been classified and routed.

3.3.2.6 Simulation results for a 2 MBit link with flooding

The rate of the outgoing link is reduced to 2 MBit/s. The settings for policer, queue manager, and link scheduler are retained according to the preceding 2 MBit link experiment in Tab. 9. The assignment of flooding traces to QoS classes is taken over from the preceding experiment with the 10 MBit link. Since the analysis of the timing does not show any new insights we focus on the maximum load values derived in this experiment.

Observation: In Tab. 13 and Tab. 14, the load values derived for a flooded 2 MBit link are listed. The former table shows the maximum load values experienced by the policer, different queue managers, and schedulers as well as by the overall system. The latter table focuses on maximum load values generated by special building blocks for priority queues and dynamic memory management as well as for payload storage. Compared with the preceding results for a

Tab. 13: Maximum load of architecture blocks for the 2 MBit/s link. The system is flooded by maximum small packets at 50 MBit/s peak rate.

	Scheduler				*Queue Manager*				*Policer*	*Overall*
Load [%]	*MD-SCFQ*	*SCFQ*	*DRR*	*SPFQ*	*CYQ*	*CYQ-enh.*	*CYQ-RED*	*YQ-Fair*		
CPU blocks										
ARM	25.9	8.9	6.4	23.7	18.7	19.6	252.0	243.9	100.8	372.0
SH4	8.4	4.0	3.5	8.3	6.0	6.6	29.2	34.5	32.4	72.8
PPC	2.3	1.6	1.4	2.3	2.4	2.6	10.1	13.5	6.8	22.0
RAM blocks										
SDRAM	10.8	1.4	0.2	10.9	54.5	54.5	90.7	163.5	37.3	204.4
SRAM	4.4	0.6	0.1	4.4	21.6	21.6	35.1	63.6	14.9	80.1

flooded 10 MBit link, the following can be observed:

- The maximum RAM and CPU load generated by schedulers roughly reduces by one third to one half. SCFQ's load comes close to the load generated by DRR.

Tab. 14: Maximum load of architecture blocks for the 2 MBit/s link. The system is flooded by small packets at 50 MBit/s peak rate. The load generated by special hardware blocks is listed. Their load values are included in the load of the respective component in Tab. 13 (besides payload storage).

Load [%]	*Payload*	*Priority queue*		*Dynamic memory allocation*
		SPFQ + YQ-Fair	*SPFQ only*	
CPU blocks				
ARM	–	39.9	1.7	5.0
SH4	–	13.5	0.7	1.5
PPC	–	4.4	0.2	0.6
RAM blocks				
SDRAM	5.4	48.3	1.6	35.7
SRAM	3.0	19.3	0.7	13.5

- The maximum load generated by the fair and RED-based queue managers increases by more than one half whereas the load for the simple queue managers remains the same.

- (Not displayed) When the peak rate of the flooding traces at the input is reduced from 50 MBit/s to 10 MBit/s, RAM and CPU load values of the overall system are again reduced by a factor of three.

  Discussion: Again, the maximum load values are experienced at the beginning of a simulation run when the policer lets incoming traffic pass at its peak rate. However, the situation shows a little difference compared with the 10 MBit settings. The initialization phase for the flooded 2 MBit link with a fair queue manager is displayed in Fig. 34. The resolution of the averaging interval has been increased from 1 ms to 0.1 ms to better show the effects. This is why the load values in Fig. 34 are slightly higher than the corresponding values in Tab. 13. Approximately after 13 ms, the queue manager must start to drop packets before the policer finally bounds the incoming traffic two milliseconds later. Within this interval when the queue manager has to cope with the incoming traffic at the peak rate, the maximum load values are experienced. This was different for the 10 MBit link where the policer was already bounding the incoming traffic when the queue manager started to drop packets.

  Conclusion for the congested 2 MBit link:

- The maximum load is generated at the beginning of a busy period when the policer lets traffic pass at its peak rate and the queue manager is forced to drop packets. The influence of a fair queue manager on the system's load can be reduced by spending more shared memory and by bounding the peak rate at the policer. If more memory is available, the policer will bound the incoming traffic

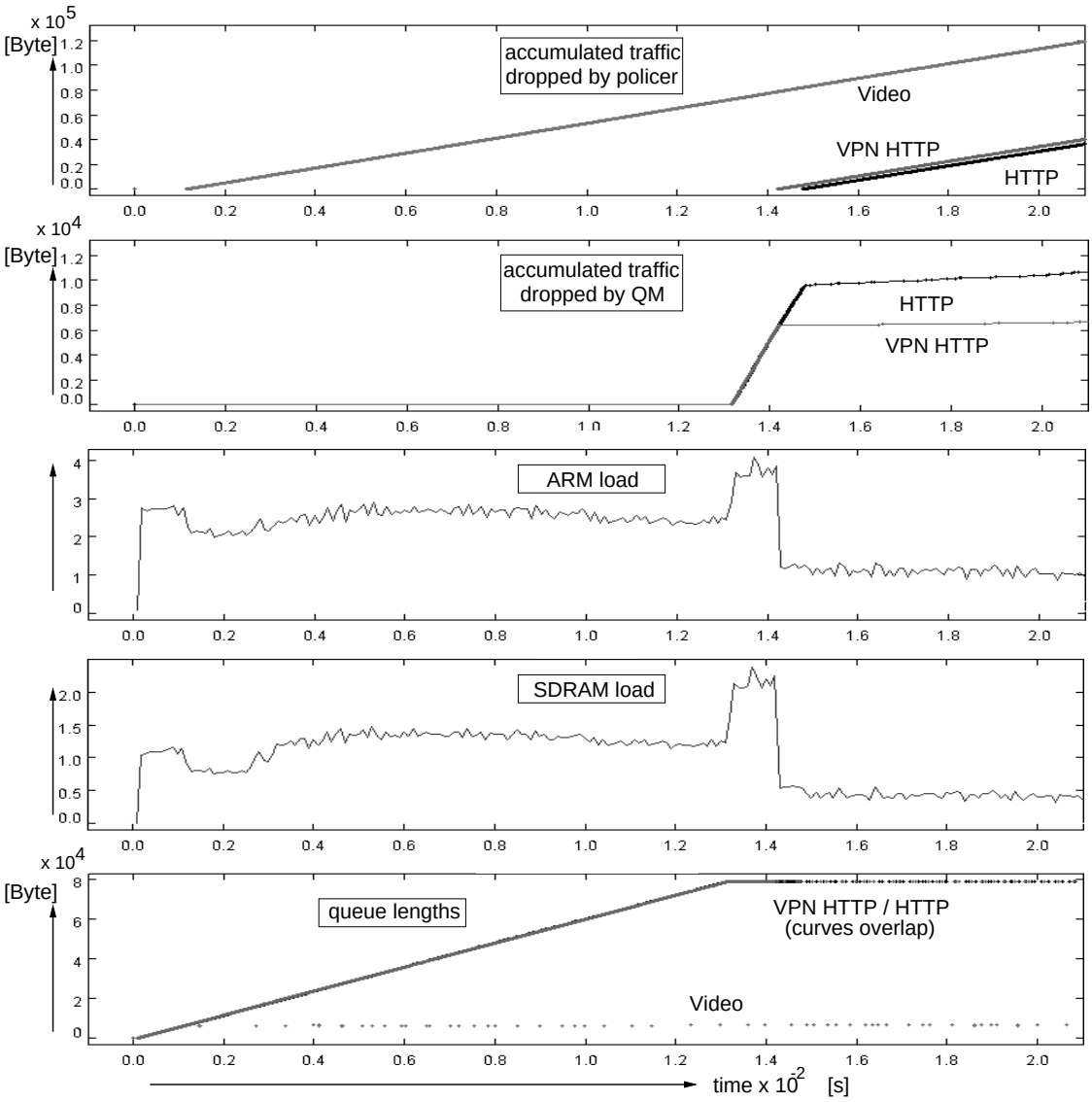

Fig. 34: Load in the initialization phase of the system. The system is flooded by small packets and the link supports a rate of 2 MBit/s. The queue manager must drop packets before the policer limits the incoming traffic.

before the queue manager starts to drop packets. Moreover, it may no longer be required to use a fair queue manager since a queue manager with a central yellow queue performs well enough for coping with rare dropping events.

- RAM and CPU load values are especially determined by the choice of the queue manager. The overhead for maintaining per-flow state for yellow traffic is considerably high when congestion appears.

3.3.2.7 Looking at fairness

The preceding experiments have looked at the timing behavior and the maximum load values of the system. This subsection focuses on the fairness properties of the overall system which is influenced by the choice of the queue manager

and the link scheduler. We use the concept of relative service as it is used for the fairness index by Golestani in Def. 7 to assess the short-time unfairness of a system. The relative service $\frac{W_i(\tau_1,\tau_2)}{r_i}$ for a flow i is defined by the served amount of flow i's traffic $W_i(\tau_1, \tau_2)$ in the interval $(\tau_1, \tau_2]$ and the reserved rate r_i.

Fairness of the link scheduler: The simulation runs use a 2 MBit/s link. Most of the settings from Tab. 9 are maintained. In order to increase the effect of unfairness, the reserved rate for the video class is reduced to 20 KBit and for the VPN HTTP class to 60 KBit. The settings for all components are adjusted accordingly. The fairness index $\mathcal{F}$ by Golestani (Def. 7 and Tab. 1) then becomes 0.78 s for SCFQ, 1.15 s for MD-SCFQ, 1.16 s for SPFQ, and 1.73 s for DRR. That means, looking at any two backlogged flows, one flow should not be ahead of the other flow by more than $\mathcal{F}$ seconds in terms of received service.

The FTP, HTTP, VPN FTP, VPN HTTP, and video classes are fed by FTP traces with characteristics as listed in Tab. 4. No congestion appears, i.e., no packets must be dropped by the queue manager. Thus, solely the link scheduler determines differences in service. In Fig. 35, the relative services for these five backlogged flows are displayed for a backlog period of approx. 0.4 s for three different link schedulers. The accounting resets afterwards because the set of backlogged flows changes. Since the plots for MD-SCFQ and SPFQ are very similar, only the plot for SPFQ is included in Fig. 35. The average slope of all curves is above one for all flows. Therefore, every flow at least receives its minimum reserved share of the link rate. The best fairness is shown by SCFQ with a variance of only 0.1 s to 0.2 s in the relative services. SPFQ's results are slightly worse than the results for SCFQ but still very close to SCFQ's fairness. DRR however shows a variance in the relative services which is three times worse than in SPFQ's case. The experienced unfairness however is far below the bounds given by Golestani's fairness index $\mathcal{F}$ for all four schedulers. Since the fairness for WFQ-based systems only depends on the maximum packet length and the smallest reservable rate, the unfairness will not necessarily increase if more flows share the link. Opposed to that, a DRR-based system will show decreased fairness in any case if more flows are backlogged since the unfairness of DRR is directly coupled with the Round-Robin frame length.

Fairness of the queue manager: In order to investigate the influence of the queue manager on the overall fairness of the system, we let the link run into congestion by reducing the available memory space and look at the behavior of different queue managers. Due to limited shared memory, for every two transmitted packets at least one packet must be dropped. In Fig. 36, the relative service for a SCFQ scheduler combined with two different queue managers is shown. The fair queue manager uses per-flow state for yellow packets and therefore drops packets in a fair manner. The queue manager with a central yellow queue does not distinguish per-flow state for yellow service but drops yellow packets in LIFO order of their arrival. Fig. 36 reveals that the choice of the queue manager does not influence the fair distribution of service among backlogged flows by the scheduler. The queue manager may only shorten or lengthen the busy period of a flow by affecting the backlog due to loss.

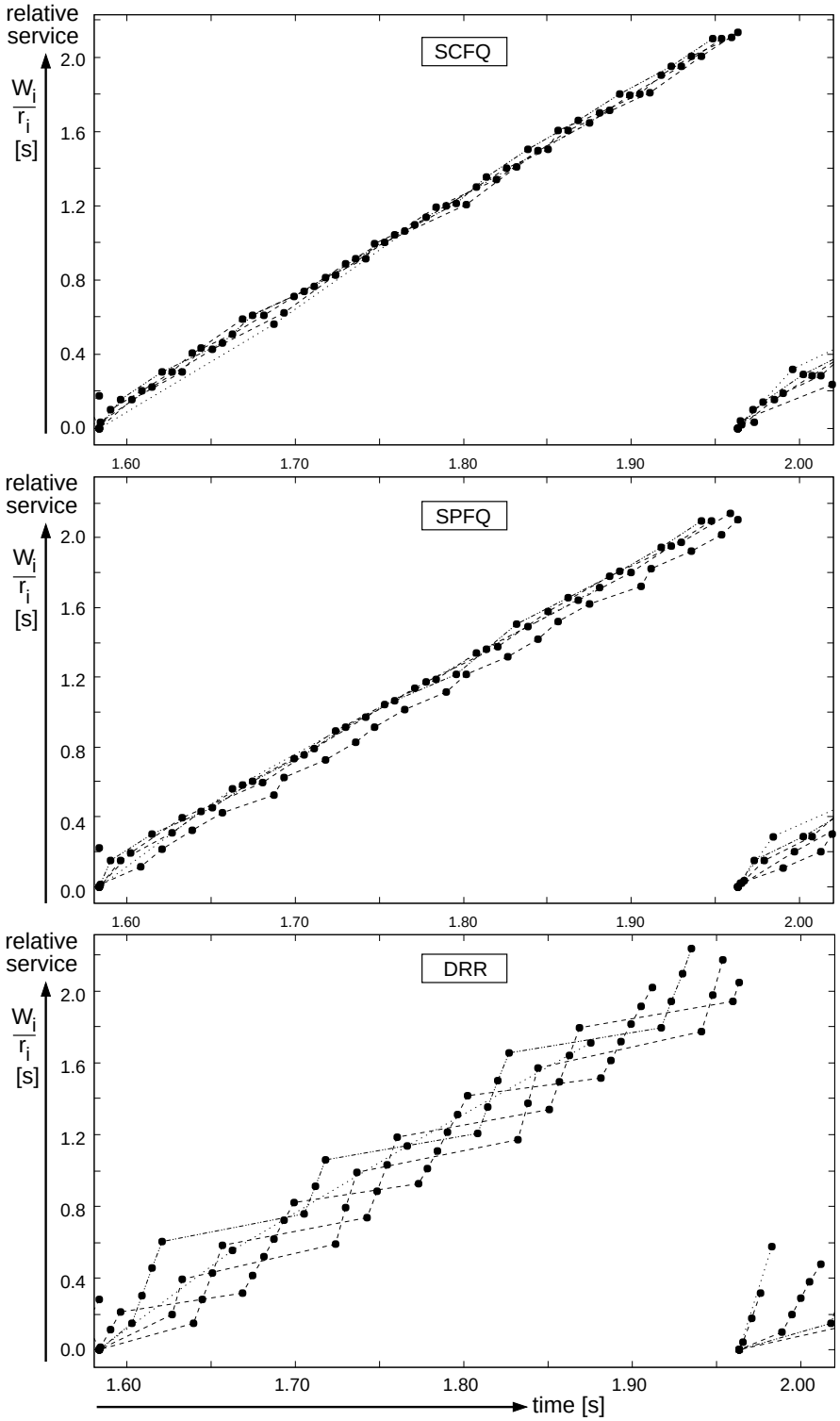

Fig. 35: Relative service for five flows backlogged for approx. 0.4 s at an uncongested 2 MBit/s link. The set of backlogged flows changes afterwards so that the accounting resets. Three different link schedulers are investigated. The relative service $\frac{W_i}{r_i}$ for a flow i is defined by the served amount of flow i's traffic W_i and the reserved rate r_i.

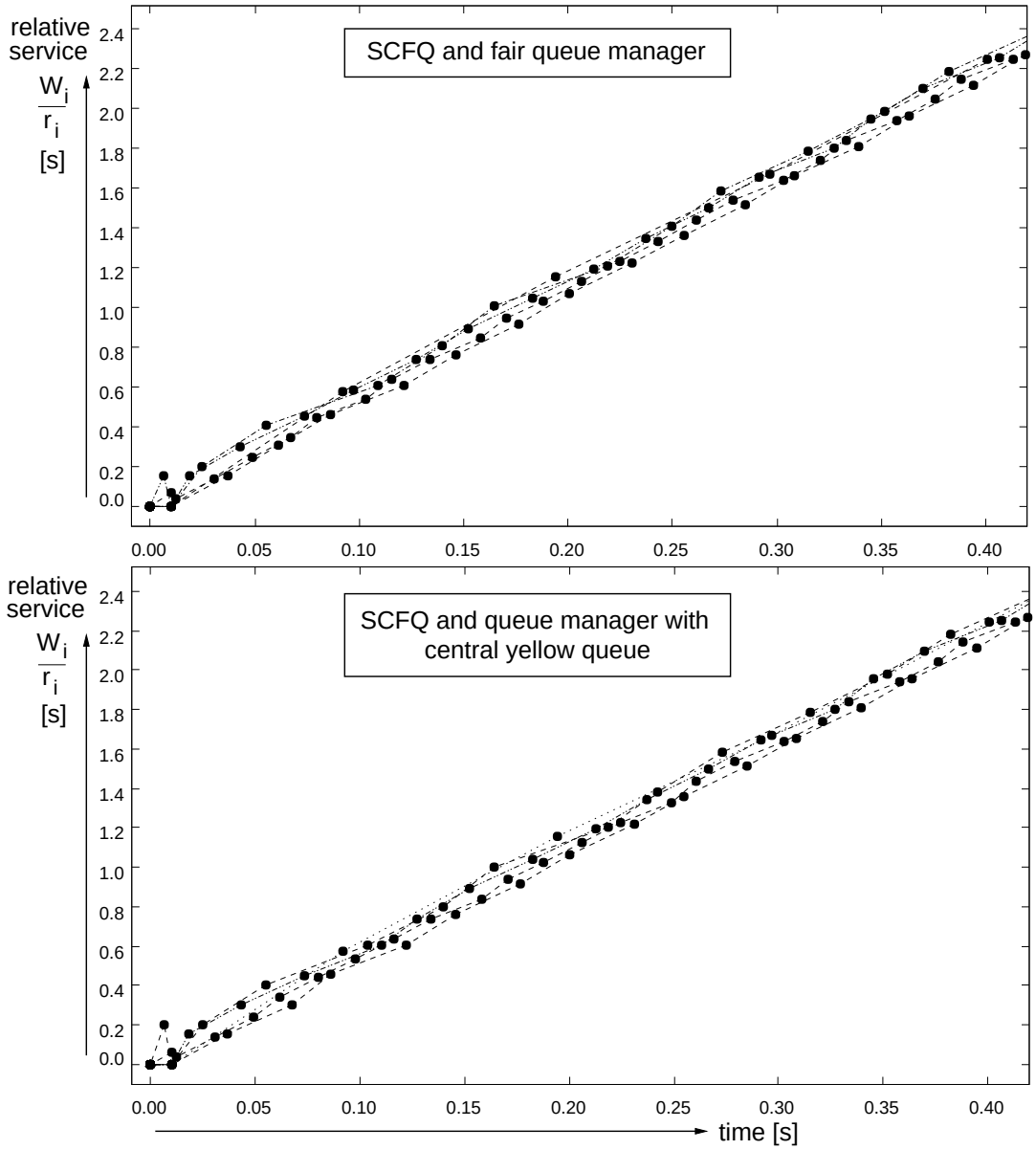

Fig. 36: Relative service for five flows backlogged for approx. 0.4 s at a congested 2 MBit/s link. Two queue managers combined with an SCFQ link scheduler are investigated. The relative service $\frac{W_i}{r_i}$ for a flow i is defined by the served amount of flow i's traffic W_i and the reserved rate r_i.

Conclusion for the fairness of a system: The fair distribution of service according to the fairness index introduced by Golestani (Def. 7) is only determined by the choice of the link scheduler. The queue manager has no impact on the characteristics of the sharing of the link service, but may only shorten the period a flow is backlogged by dropping packets to a greater extent. The choice of the queue manager in turn defines the fair push-out of already stored packets and thus influences the loss rate experienced by a flow. However, a link scheduler that does not provide the tightest delay bounds will potentially increase the

probability to experience loss if the amount of shared memory is not increased accordingly (recall Fig. 32).

3.3.3 Conclusion of the design space exploration

The description of the simulation results is concluded by replying to the questions which have been put at the beginning of the discussion. For our system configuration and the number and kind of supported QoS classes we can state:

- *QoS performance:* Since our provision of Quality of Service (QoS) for guaranteed (green) service is based on resource reservation and schedulability tests, the system is always capable of keeping delay guarantees without experiencing any loss. The choice of a scheduler and a queue manager thus influences the number and the type of different demands of flows that can be supported as well as the way surplus resources are shared. A Deficit Round-Robin (DRR) scheduler cannot support the same variety of delay guarantees as Weighted Fair Queueing (WFQ)-based schedulers can. Low bandwidth flows receive considerably worse service in terms of delay in a DRR-based system than in a WFQ-based system.

- *Distribution of CPU and RAM load:* The CPU load will be balanced among policer, queue manager, and scheduler if a WFQ-based scheduler is chosen together with a simple queue manager that only uses a central queue for yellow-marked traffic. The CPU load will be dominated by the queue manager if a fair queue manager is employed which distinguishes per-flow state for yellow-marked traffic. The share of the CPU load caused by the link scheduler can considerably be reduced by using a Deficit Round-Robin based scheduler. The load of the RAM is determined to a great extent by the queue manager.

- *Scaling of the load with the supported link rate:* The maximum load experienced by the policer and the queue manager does not depend on the supported link rate but rather on the characteristics of the incoming traffic. The maximum load caused by the link scheduler may potentially double for CPU and RAM when the supported link speed is increased from 2 MBit/s by a factor of five. Opposed to that, the load caused by the overall system triples when the peak rate of incoming greedy traffic is increased by a factor of five. That means, the rate of the access link rather plays a secondary role for the determination of the worst-case load of the system.

- *Worst-case scenario:* The highest load is generated at the beginning of a busy period when the policer lets traffic pass at its peak rate and the queue manager is forced to drop packets. The effect becomes stronger with smaller packet sizes.

- *Queue manager fairness:* The queue manager is responsible for the fair push-out of packets from the shared memory during times of congestion and thus decides the experienced loss rate of a flow. The dropping behavior of the queue manager however does not affect the properties of the fair sharing of service defined by the link scheduler. It is not worthwhile to afford a per-flow distinction

for yellow-marked traffic since the overall role of the queue manager in coping with congestion can be reduced by getting the system more shared memory. That means, the overhead for avoiding congestion by the policer is strictly less than the overhead of recovering from congestion by the queue manager.

- *DRR vs. WFQ timing behavior comparison:* Although only a small number of QoS classes are supported, the delay penalty imposed by DRR can be considerably large and in addition increases the likelihood of loss experienced by a flow. Therefore, our access link scenario already shows the usefulness of a WFQ-based scheduler for preserving the QoS of a small number of classes.

- *RED vs. Fair queue manager complexity comparison:* The overhead for maintaining the average queue length to apply Random Early Detection (RED [54]) congestion avoidance is almost as high as for supporting per-flow distinction for congestion recovery. Since congestion avoidance is performed by the policer anyway, implementing RED is therefore not worthwhile from a complexity point of view.

- *Performance bottle-necks:* In the current state of development, the policer and queue managers based on RED or per-flow distinction for yellow traffic cannot be implemented on a simple ARM CPU core. If a CPU with a floating-point unit cannot be used but the support for per-flow distinction or RED is mandatory, the complexity must be decreased by reducing the size and kind of the data type or by applying heuristics to determine the longest queue and the average queue length respectively. If a CPU with a floating-point unit was used, the whole system could be implemented on a single CPU core. Due to its simplicity, the policer anyway is a candidate for a hardware implementation with programmable counters.

  The same statements accordingly apply to the required RAM resources. The more advanced queue managers cannot be implemented based on SDRAM technology. If however SRAM technology was utilized, parameters and variables for all components of the system could be mapped onto a single SRAM area. If a simple queue manager is implemented and SDRAM is used for parameters and variables, separate memories must be afforded for the queue manager and the remaining components of the system.

  The results for certain building blocks to implement priority queues and dynamic memory allocation show that their acceleration by special hardware blocks would not affect the overall system speed sufficiently to be a worthwhile investment since their share in the overall load is too low.

- *Influence of high priority voice traffic:* Voice traffic should carefully be limited. A tolerable share seems to be 10% of the link bandwidth. A higher share potentially jeopardizes the teamwork of policer, queue manager, and link scheduler to provide a reliable service without bursts of lost or transmitted packets.

- *Trading timing behavior off for complexity and fairness:* A configuration with Deficit Round-Robin (DRR) and a queue manager with a central queue for yellow traffic clearly shows the lowest complexity but the worst timing and fairness. For the 2 MBit link however, Self-Clocked Fair Queueing (SCFQ) already comes close to DRR in terms of caused load. Due to the better timing and fairness behavior of SCFQ, it should be preferred as link scheduler for that system. If guarantees for tight deadlines are a major concern, a Weighted Fair Queueing (WFQ)-based scheduler is mandatory. Since the timing behavior of Starting Potential-based Fair Queueing (SPFQ) and Minimum Delay Self-Clocked Fair Queueing (MD-SCFQ) is very similar, one should choose the service discipline with lower complexity. For our configuration that would be SPFQ. Moreover, MD-SCFQ and SPFQ's difference in short-time fairness compared with SCFQ is probably of no practical significance. As discussed before, it is not worthwhile to implement a fair queue manager since its importance can be decreased by spending more payload memory and by adjusting the policer. In conclusion, we encourage the use of a WFQ-based link scheduler although the number of classes to support is rather small. The best timing for our purposes is then provided by SPFQ whereas the lowest complexity is offered by SCFQ. Both service disciplines provide almost perfect fair sharing of surplus bandwidth.

As a result we suggest the following System-on-a-Chip solution in Fig. 37 to implement the system consisting of a policer, a queue manager, and a WFQ-based link scheduler in its current form without any further optimizations of algorithms or data types. We use a central CPU core that must have a dedicated floating-point unit, two separate on-chip memory areas which may be based on different technologies, and an off-chip payload memory. If the three

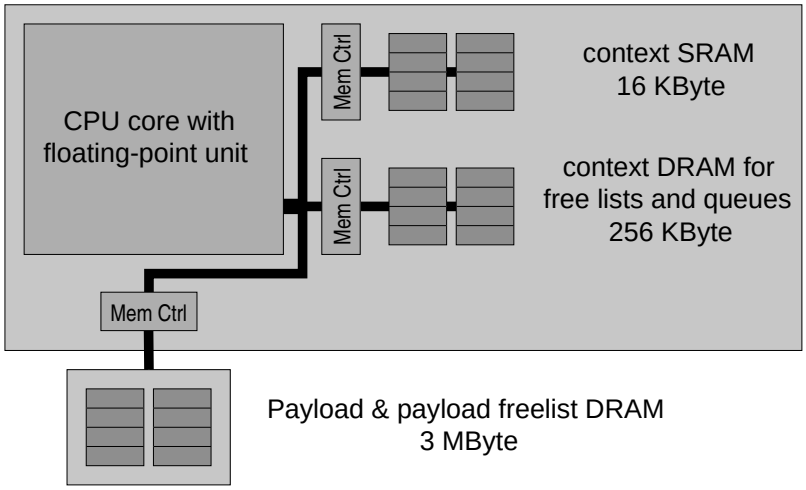

Fig. 37: Suggested implementation of the packet processor based on the assumption that algorithms and data types are not optimized further.

required memory controllers are not integrated into the CPU, they will share a common front-side bus to the CPU as sketched in the figure. This is suitable

since the load values derived from the simulations have been added to dimension the RAM resources. That means, all accesses of the RAMs can be performed strictly sequentially without overloading the architecture. As discussed in subsection 3.3.2.2, the performance models of algorithms cannot provide any precise estimates of the required code memory but we believe that an additional 64 KByte memory segment should be more than enough to implement policing, queue management, and link scheduling. Whether this code segment can be mapped onto payload or context memory cannot be answered at this place since the required throughput is not known.

3.4 Related Work

Architectures of current network processors: The number and kind of tasks supported by network processors of different manufacturers vary greatly. The only common property seems to be that network processors are targeted on the acceleration of packet processing tasks somewhere above the physical layer of a network so that they can be distinguished from integrated switching and routing solutions. However, popular "stars" in the universe of packet processing tasks can be found which include header parsing, classification and routing, traffic reshaping as well as queuing. Packet scheduling often is not part of a network processor. Depending on the field of application and the place in the network where the processor will be used a diversity of architectures is available.

- *Low bandwidth communications without QoS distinction in hardware:* In the same way a general CPU core becomes an embedded CPU by adding special purpose blocks and a number of I/O interfaces to cope with sensor signals and interrupts, a general CPU is converted to a network processor by adding hardware blocks, for instance, that take over Ethernet medium access as well as packet segmentation and reassembly. A network processor can be considered to be a special case of an embedded processor. Examples are NetSilicon's NET processor family, Virata's communication processors, Intel's IXP225 and Conexant's CX82100 which are all based on ARM CPU cores. An interesting technology platform for the implementation of network processors with some dedicated functions may be the combination of a CPU core, a memory controller, and an array of reconfigurable logic or processing elements offered by Tricend's A7 or Chameleon Systems' CS2000 System-on-a-Chip designs.

- *Medium bandwidth – fast Ethernet to rates of one GBit/s:* The simplest approach is to use a faster general CPU core. Galileo Technology's Discovery product line offers a variety of MIPS- or PowerPC-based network processors. Another examples is Hitachi's SH7615. The CPU cores can be augmented with chips sets – ASICs that accelerate special network-related functions. MMC Networks' AnyFlow line of chip sets can speed up classification, routing, per-flow queuing, and scheduling. ASICs specializing on classification and partly

on header parsing and modification are PMC-Sierra's PM2329 ClassiPI, Intel's IXE100, and Solidum's PAX.port 1100. Chips that are solely responsible for encryption are offered by Chrysalis-ITS and Broadcom.

- *High bandwidth:* In order to sustain bandwidths beyond one GBit/s the exploitation of parallelism becomes more and more important. The following developments can be observed:
  - *Multi-processing:* Several CPU cores are employed that can be grouped in different ways. A CPU may exclusively be allocated to a particular packet processing task. Several CPUs then form a pipeline and packets are handed from processor to processor. EZchip Technologies' TOPcore uses this approach. Four processors in a chain support classification and policing. Scalability is preserved by extending the system in a superscalar way. Another approach is to assign a CPU to an outgoing link or to a set of flows. In this case, the CPU is responsible to perform all packet processing tasks for that link or that set of flows. This binding of tasks to processors is especially attractive since the communication between processors is minimized. Data dependencies among packet processing tasks are usually caused by packets from the same flow or packets sharing an outgoing link. CPort's C-5 Digital Communications Processor (DCP) uses 16 MIPS-based CPU cores. These CPUs can be assigned to flows or whole links so that they perform all tasks that come up with the packets. Alternatively, the CPUs can be arranged as a pipeline of processors. Intel's IXP1200 uses six RISC CPU cores – so-called microengines – that can individually be assigned to tasks. A superordinate StrongARM CPU core is responsible for managing the workload of the microengines and can be used for processing as well. In IBM's PowerNP NP4GS3 one can find 16 so-called picocode engines that are optimized for 32 Bit bit-wise ALU operations. Packets are individually assigned to processors by a dispatch unit. There additionally is a superordinate PowerPC CPU core.
  - *RAM diversity:* All chips of this class have in common that they use a variety of RAM areas and types. The CPU cores have their own local memory to store parameters and variables. Larger data sets such as routing tables may be stored in off-chip SRAM. Payload is redirected to DRAM-based memory. That means, a packet processor for high bandwidths usually employs several types of memory and corresponding controllers.
  - *Several thread contexts:* Since the RAM resources usually are the performance bottle-neck of a network processor, the CPUs support fast context switches and multiple threads running quasi-concurrently by large register sets and several program counters. In this way, the penalty introduced by threads waiting for RAM resources can partly be hidden by delegating the CPU to other threads. In Intel's IXP1200, a microengine supports four threads. A channel processor in CPort's DCP also supports four threads. A picocode engine in IBM's PowerNP maintains up to two threads.

– *Special hardware units:* Although the discussed network processors offer a high number of CPU cores, they additionally employ dedicated processing units that are optimized for certain tasks and shared by the CPU cores. CPort's C-5 uses special hardware blocks for queuing and forwarding. Intel's IXP1200 employs a special unit to generate hash keys. Almost all general packet processing tasks are supported by hardware blocks in IBM's PowerNP including parsing, classification and forwarding, policing, queuing, and link scheduling.

In the end, all network processors use a general-purpose CPU to some extent. This allows to adapt to changes of protocol and other networking standards. Mature software development tools can partly be reused which not only shortens time-to-market but also increases time-in-market periods.

Issues in current research: Only few papers about network processors are available so far. Most of the papers are restricted to a comparison and analysis of architecture or technology issues without any statements about practical performance. A survey of currently available network processors and their application areas can be found in [60, 38]. In [61], CPort's Digital Communications processor is introduced. Issues of network processors for fast backbones are discussed in [26]. In particular, a heuristic is presented which can be used to balance the load of CPU cores within a multi-processor system in which flows are assigned to CPUs and a CPU performs all packet processing tasks required for that flow. In [118], beginning with an ARM CPU core, new instructions and extensions to the CPU are introduced based on a static analysis of tasks a network processor has to perform. The new instructions support unaligned data accesses and bit-wise operations to better cope with streams. The execution of loops and branches is accelerated to achieve higher performance for control-driven tasks. A fast task switch is implemented for at most four tasks. The processor is targeted on low bandwidth applications of some MBytes/s. The paper does not present any performance investigations. In [37], a MIPS-based CPU is enhanced with fast interrupt handling, with variable-size data movements from interfaces to the RAM, and with zero-overhead context switches. The usefulness of the approach is shown by simulation of a segmentation and reassembly benchmark. A technology design study is presented in [106] to show how dynamically reconfigurable logic can be used to implement certain functionality of a network processor to find a reasonable trade-off between hardware utilization, speed, and adaptability to particular processing needs of flows. The performance evaluation of Intel's IXP1200 network processor in [144] gives an insight into the problem to make full use of a network multiprocessor that contains shared resources. The overhead of dynamically allocating tasks to microengines has been avoided by applying a static allocation scheme. In this way however, microengines can only be utilized to two third of their full capacity. Moreover, the evaluation platform will be limited by RAM speed if the system is flooded by small packets. The influence of the CPU architecture on the processing speed of software implementations for forwarding, encryption, and

authentication is investigated in [39]. In particular, it is pointed out how superscalar architectures, multi-threaded extensions, and multi-processor setups as well as combinations of these features have an impact on the packet processing speed of the system. The simulation results underpin why multi-threaded multi-processor systems are so popular for high bandwidth network processors.
Standardization in progress: Different committees have been founded to facilitate the design of a switch or a router based on network processors from different manufacturers and to enable the reuse of hard- and software components. The Common Switch Interface (CSIX) forum is responsible for the definition of hardware interfaces between network processors and a switch fabric as well as between different network processors. The Common Programming Interface (CPIX) forum's aim is to define standardized application programming interfaces (APIs) between network processors and other hardware and software entities. Finally, the Network Processing Forum (NPF) wants to define standardized benchmarks and test cases to ease objective comparisons. The different working groups can be found at [1]. Independently of the NPF, a first suggestion for a set of software kernels that can be used as a benchmark for network processors used at access networks is presented in [163].

3.5 Summary

The concept of a multi-provider/multi-service access network has been introduced in this chapter. We have suggested a premium service with guaranteed lossless transmission and delay bounds for individual flows or flow aggregates. The service will gracefully be degraded by weakening delay bounds and introducing loss if the customer's traffic exceeds an agreed traffic profile for premium service. In this way, the advantages of IntServ and DiffServ have been combined. We have discussed how the suggested service model influences the choice of algorithms for policing, queue management and link scheduling.

A thorough design space exploration for algorithms and architectures aimed at network processors supporting multi-service access networks has been performed. The main results from exhaustive simulation runs can be summarized as follows:

- Even if a small number of classes are supported, Weighted Fair Queueing-based systems will show significantly better delay properties than Deficit Round-Robin-based systems.

- Worst-case load is generated in the initialization phase of the system after an idle period when traffic passes the policer at its peak rate.

- The CPU load is rather balanced among policer, queue manager, and scheduler whereas the RAM load is determined by the queue manager for the most part.

- Load peaks will be decreased if packets are preventively discarded at arrival at the policer rather than dropped by the queue manager to cope with congestion.
- An implementation of the network processor without any further optimization of the algorithms would only require a single CPU core but several memory areas. The load generated by priority queues and dynamic memory allocation is too small to justify a dedicated hardware implementation.

For the first time, the resource utilization of different hardware configurations has been evaluated concurrently with an investigation of the interplay of various combinations of policing, queue management, and link scheduling to preserve Quality of Service. As a result, reasonable trade-offs among complexity, delay, and fairness have been explored for network processors aimed at access networks. A preliminary study how the presented multi-service access node can be augmented with support for a line-like shared-medium distribution network can be found in [69]. The QoS distinction mechanism described therein is used as a case-study in [154].

4

Exploitation of RAM resources

The potential speedup for quality of service networking support by using a dedicated network processor mainly depends on two factors: the acceleration of network-specific computations and the exploitation of available RAM resources. The computation part is discussed in Chapter 3. The influence of the RAM part is the main focus of this chapter. A network processor has to manage a diversity of data structures such as packets that must be buffered, context information that is used to decide to which outgoing link a packet should be forwarded, and quality of service-specific parameters and values. For most of the data structures it is mandatory that a network processor is able to access them fast enough to process packets at "wire speed" of the supported network. This is why almost all commercially available network processors have to employ a multitude of different RAM areas and types – as it is shown in the related work section of Chapter 3. Moreover, the characteristics of packet processing tasks imply that caches either are prohibitive or cannot be exploited due to the lack of access locality, see Chapter 2. Therefore, great significance must be attached to the exploitation of RAM resources and this chapter is devoted to this topic.

This chapter will answer the following questions:

- There is a diversity of different RAM types. What are their specific characteristics and for which application areas are they well suited?
- What is the impact of the memory controller on the performance of the overall system? A memory controller is placed between a computing core and RAMs and can thus take advantage of different operating modes of the memory chips to shorten memory access schedules. Its complexity is manageable so that application-specific controllers could be integrated into a network processor.

- What does the performance of a current DRAM influence more, the throughput of the interface to a processor or the delay properties of the underlying memory core? Is it therefore worthwhile to use a RAM with a high throughput interface?

Two models of different resolution are elaborated to evaluate RAMs and memory controllers in detail. First, a detailed model of a controller and a dynamic RAM chip is developed to document, analyze, and optimize the interplay of controller and memory for typical memory access patterns. This step is mandatory since the architecture and the functionality – such as state machines – are not documented by manufacturers. The memory controller is therefore reverse-engineered from RAM data sheets. The insights gained by the refinement of the model are then used to combine an abstract performance model of the controller and suitable RAMs with a mature CPU simulator so that whole benchmarks can be applied to evaluate the performance of a memory subsystem. We are hence able to show for the first time, how heavily the overall computing performance of an embedded system actually depends on the exploitation of RAM resources by a memory controller. Moreover, by augmenting the controller with more resources than there are in currently available controllers we assess the potential to make full use of recently emerged RAM features.

Why RAMs are a potential performance bottle-neck of network processors is analyzed in the next section by describing the architecture and properties of dynamic and static memories. Section 4.2 emphasizes why the memory controller is the key component to speed up memory accesses. Section 4.3 presents a detailed performance model of a memory subsystem described by a visual formalism to analyze and optimize the interplay of controller and memory. Section 4.4 continues with a design space exploration of the performance of a memory subsystem by varying the type of used DRAM, the features of the memory controller, and the application which is processed by the memory subsystem. The chapter concludes with a discussion of related work.

4.1 Performance bottle-neck of RAMs

Memory chips have become the main performance bottle-neck of computing systems [18, 167, 24]. Chip manufacturers try to bypass the weak performance of the asynchronous memory core by designing complex memory interfaces which isolate the behavior of the slow memory core from the fast interconnections between the memory chip and the memory controller. This leads to a variety of synchronous interface implementations which essentially are based on the same memory core technology and functionality.

In this section it is shown why the delay caused by read and write accesses of dynamic RAMs (DRAMs) depends on the state of the RAM, the placement of data within the RAM, and the order of the accesses. Contrary to that, access delays caused by static RAMs (SRAMs) may only depend on the type of access

– read or write. This section is concluded by an overview of available RAM types. A more thorough discussion can be found in [66]. Only the most recent RAMs with a synchronous interface are considered in this section.

4.1.1 Organization of DRAMs

A certain amount of data, typically four to 32 Bit, forms the smallest accessible piece of information that can be addressed. Memory cells are arranged in a two-dimensional array. An arbitrary access therefore is a two-step process. The pin count of a RAM chip is reduced by successively transferring the row and the column addresses to the RAM. In this manner, a whole row within a memory array is selected first and a column is chosen thereafter. In the context of DRAM organization, a row of an internal memory array is often called a memory *page*. After a row has been addressed, the contents of that row are held by the so-called *sense amplifiers*. Thus, the sense-amplifiers can be seen as an internal single row cache of the corresponding memory array. Accesses to that particular row are fast and just need the column access time. Accessing another row however is slow, since the current row in the sense-amplifiers must always be precharged before another row of the array can be addressed. A memory array together with decoders and sense amplifiers is usually called a *memory bank*. A RAM chip may consist of several memory banks. On the one hand, each memory bank within a memory chip has its own row of sense amplifiers. On the other hand, the banks within the same chip must share input and output pins and buffers. Moreover, recent DRAMs have pipelined synchronous interfaces which isolate the memory arrays from memory bus signals. Read and write accesses to consecutive addresses are thus usually performed in burst operation mode.

A detailed description of the organization of RAMs with the main focus on electronical properties and semiconductor structures as well as the development history of dynamic RAMs can be found in [5, 45].

4.1.1.1 DRAM operation

- ***Read accesses:*** If the CPU wants to read the contents at a particular address in the RAM, the memory controller will at first have to transfer the bank number and the row address of that location to the RAM. The RAM chip now decodes the row address and transfers the information of that row to the sense amplifiers. This first phase of a read access is often called the *activation* of a memory row. In order to select a particular column entry within the activated row, the memory controller now has to supply the according column address. After decoding, the RAM is able to drive the data bus with the contents of the sense amplifiers for that column entry. Subsequent column entries will be transferred through the data bus during consecutive clock cycles without any further control signals of the memory controller if a burst read instruction is used. The column address is automatically incremented by the RAM. An example for a burst read operation of eight data words is sketched in Fig. 38.

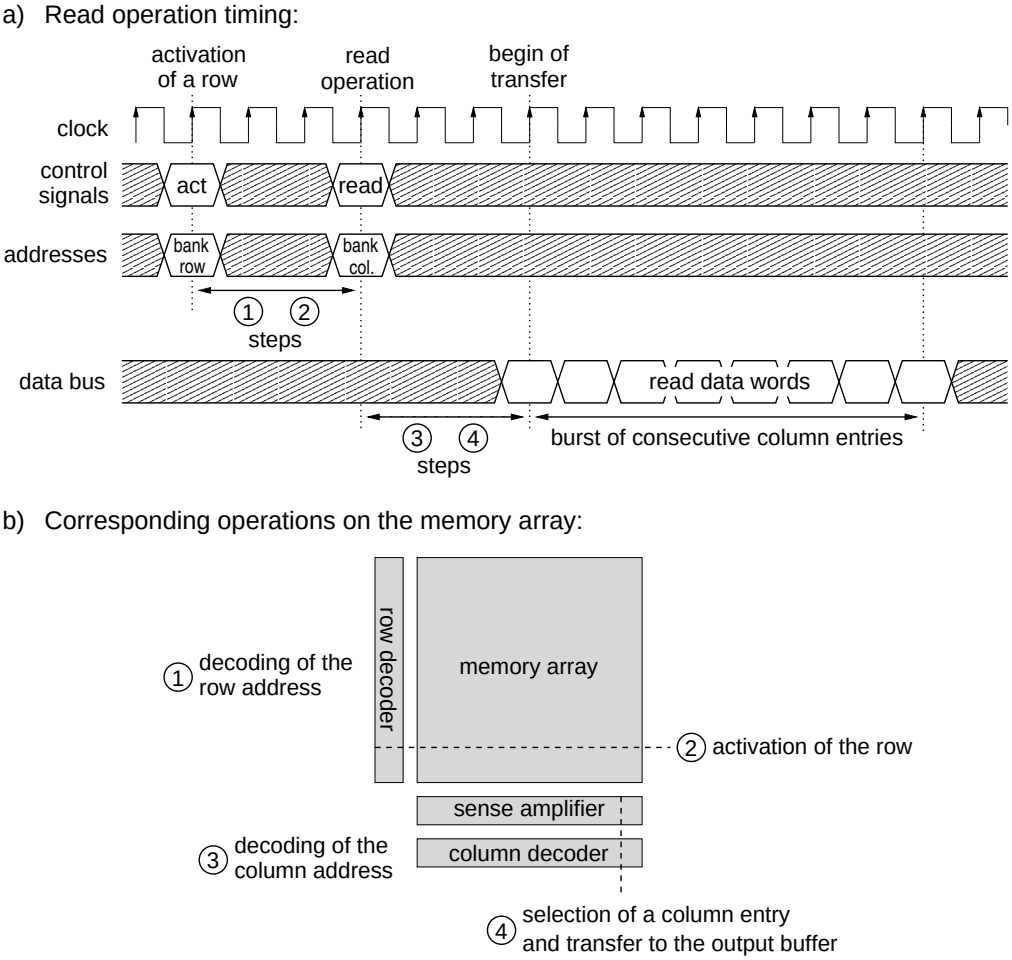

Fig. 38: General functionality of a read operation: timing a), functionality b).

- ***Write accesses:*** At the beginning of a write access, a row of the memory array must be activated in the same way as in the read access case. That is, with the help of a bank number and a row address, the information in the corresponding row is transferred to the sense amplifiers for further processing. Then, the memory controller has to transmit a column address and the data word which will be written to a particular column entry of the activated row. A write access may also use a burst write instruction to simplify write accesses to subsequent column entries. The memory controller must just drive the data pins of the RAM with different data words in consecutive clock cycles. An example for a burst write operation of eight data words is sketched in Fig. 39.

- ***Precharging:*** Accesses to memory cells of a dynamic RAM through the sense amplifiers are destructive, i.e., the charge is lost and cannot be reconstructed by the cell itself. Hence, a precharge operation is needed to reconstruct the cell contents based on the current information that is held by the sense amplifiers. The memory controller must initiate the precharge of a memory bank before it is allowed to activate another row of the same memory array. Some of the

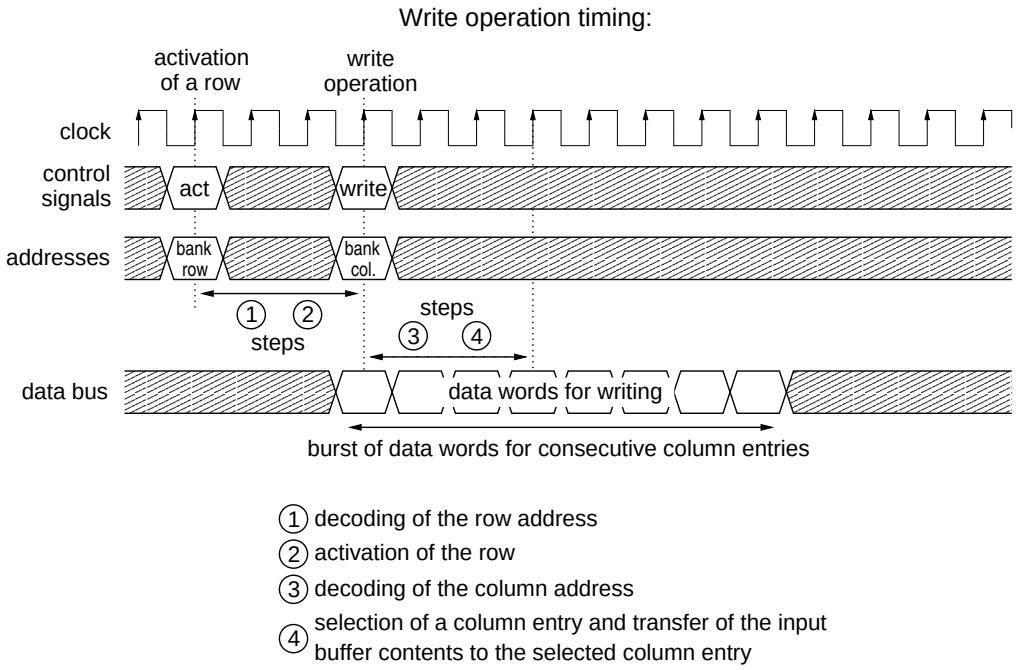

Fig. 39: General functionality of a write operation.

delay introduced by the precharge operation can be hidden if a read operation is performed before the precharge. The precharge of the corresponding memory array may be performed concurrently to the read out of the output buffer because the contents of the sense amplifiers are no longer needed.

- ***Refreshing:*** A dynamic RAM slowly looses the charge of the memory cells due to leakage effects. This is why the charge of the cells must periodically be restored. This process is called *refresh* and implemented by an activation and a following precharge of each row of all memory arrays in the RAM. The whole device can be refreshed at once. Alternatively, one can refresh only a single row by time in smaller intervals. In the latter way, the delay penalty for refreshing the device is distributed over the whole refresh interval.

4.1.1.2 DRAM Timing

Around 30 to 40 timing parameters are necessary to describe the exact timing behavior of a RAM chip. Only some of them are pointed out at this place which will be sufficient to model the behavior of a RAM if it is properly controlled, for instance, without breaking an instruction by issuing another one too early.

- t_{RCD}: This is the time it takes to activate a row of an idle memory bank. That means, the memory bank is in a precharged state and not subject to be refreshed within some few clock cycles before the activation. After a row address has been decoded, the sense amplifiers are filled with data from the memory array.

- t_{RP}: This time is needed to precharge an active row. The charge of the memory cells of a row is restored according to the information held by the sense am-

plifiers. The memory bank is in an idle state after the precharge operation has finished.

- t_{RAS}: This is the minimal period an activated row must be kept activated before it may be precharged. This parameter is measured from the beginning of the activate instruction to the beginning of the precharge operation for the same row. That is, t_{RAS} includes t_{RCD}.

- t_{CAS}: This parameter describes the so-called *column address strobe delay* or CAS latency (CL). This is the time between issuing a read instruction on the control bus and the appearance of the first data item on the output pins.

- t_{DPL}: This delay must be spent between the transfer of the last data item of a burst write and a following precharge operation in the same bank. This delay assures that the information which is held by the sense amplifiers is up to date and that the array is precharged accordingly.

- t_{REF}: The maximal refresh interval of the whole memory chip is specified by this value. The RAM chip must completely be refreshed at least once within an interval of length t_{REF}.

- t_{WAR}: This is the minimal write-after-read operation delay. This time interval must be spent between the transmission of the last data word of a read operation on the data bus and the transfer of a following write instruction on the control bus. This delay may be necessary because the flow direction of data words changes on the bus. By violating this delay requirement, there may be the possibility that data read out from the sense amplifiers collides with data which is already latched into the RAM chip for writing.

- *burst length:* The length of a burst of read or write accesses is specified by this parameter. Typical burst lengths are four column entries or a full page burst. This length is the number of data word items – i.e., the number of column entries in the sense amplifier row – that are transferred consecutively without the need of providing new column addresses by the memory controller.

The timing parameters are illustrated in Fig. 40.

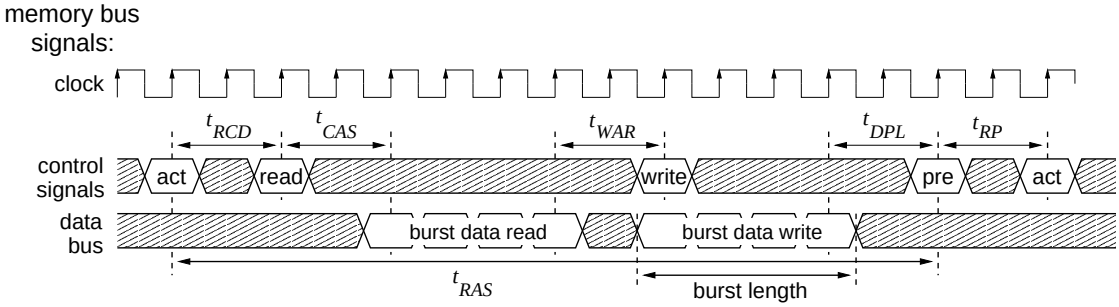

Fig. 40: Exemplary write after read access timing.

4.1.2 Organization of SRAMs

Synchronous SRAMs use the same operation modes as synchronous DRAMs. Read and write instructions may be used in burst mode, the synchronous interface is pipelined and thus allows to issue a memory instruction in every clock cycle. Since SRAMs are optimized for speed as their main application area is the implementation of caches, the address of a data item is not split into row and column addresses but fully specified in one clock cycle. The pin count is increased but an arbitrary data item can be read out with a delay of just one or two clock cycles independently of the address and the state of the SRAM. Assuming a read access delay of two clock cycles a DRAM with equivalent timing ought to have the timing parameters $t_{RCD} = 0$, $t_{CAS} = 2$ cycles, $t_{RAS} = 0$, and $t_{RP} = 0$. Due to SRAM technology, a refresh operation is not necessary and there is no limiting minimal row active time parameter that must be kept. On the other hand, the worst-case power dissipation of an SRAM in operation is higher than for any DRAM type. SRAMs are roughly one order of magnitude more expensive than DRAMs with the same capacity since SRAMs have a higher pin count and need more silicon area.

4.1.3 Available synchronous RAM types

4.1.3.1 Synchronous DRAMs

SDRAM: SDRAM currently is the most often used synchronous DRAM type. Especially, the RAM modules according to the PC 100/ PC 133 SDRAM specification [88] are popular. SDRAMs are available on standardized modules with a fixed data bus width on which several memory chips are mounted. The most widespread modules are Dual In-line Memory Modules [92] (DIMMs) that offer a 64 Bit wide memory bus. SDRAMs usually employ two to four internal memory banks. The overhead for refreshing an SDRAM can be kept below 1% of the run-time. A variety of modified SDRAMs is available, including:

- *Virtual-Channel (VC) SDRAM:* The enhancement introduced with VC-SDRAMs consists of a number of small SRAM cache areas called channels. The channels have a size of a quarter of a row called a segment. The contents of a row's segment can be transferred into an arbitrary channel. All read and write operations perform on channel contents. Since the memory arrays and the channels can operate concurrently, activation, precharge, and refresh penalties can completely be hidden. However, the memory controller has to take care of the data consistency of the sense amplifiers and channel contents.

- *Enhanced SDRAM:* The performance of ESDRAMs is increased by coupling each memory bank with an additional full-row SRAM cache line. These caches can be used concurrently to the basic sense amplifiers. However, the SRAM row caches can only be utilized for read accesses. Read access overheads and refresh operations may completely be hidden as long as read operations can be fulfilled by the SRAM cache contents. However, write operations always perform on the sense amplifiers.

- *Synchronous Graphic RAM (SGRAM):* SGRAMs are smaller and thus faster SDRAMs with additional functionality that especially supports basic graphic operations. SGRAMs additionally have two special registers, a bit mask register and a color register. The bit mask register is used to individually mask bits on the data bus during a write operation. In block write mode, the color register may be copied into eight consecutive column entries of a sense amplifier row in just one clock cycle. The data bus width per chip of 32 Bit is larger than the bus width of ordinary SDRAMs (four to 16 Bit).

- *Double Data Rate (DDR) SDRAM and SGRAM:* DDR-SDRAMs and DDR-SGRAMs are high throughput variants of their single data rate (SDR) counterparts. Contrary to SDRAMs and SGRAMs, DDR-RAMs use both the rising and the falling edge of the clock for data transfers in order to double the potential throughput of the input-/ output interface.

Direct Rambus DRAM: Direct Rambus DRAM (RDRAM) is a memory specification developed by Rambus Inc. The Rambus approach for improved memory utilization is to use narrow buses, so-called channels, at very high clock rates. In traditional SDRAM based main memory systems, several chips are needed to fit the width of the memory bus. Thus, these chips cannot be controlled concurrently. A single RDRAM chip, however, spans the whole Rambus channel so that two RDRAM chips can already be controlled concurrently. The Rambus memory bus has a width of only 16 Bit. The transmission of all kinds of information needs four clock cycles for completion. Since even control signals need several clock cycles for transmission, RDRAMs are often referred to as DRAMs which use packets for communication. RDRAMs have write buffers. The memory controller must take care of the data consistency between the write buffers and the sense amplifier contents. RDRAMs are also available on memory modules, so-called RIMMs specified by Rambus Inc. Several RIMMs are interconnected in a serial fashion.

4.1.3.2 Synchronous SRAMs

On the one hand, a *pipelined burst SRAM* (PBSRAM) usually needs two clock cycles to read out the contents at an arbitrary address after the read instruction has been issued. On the other hand, write data must instantaneously be supplied with the write instruction. Thus, the data bus cannot completely be exploited if a read access follows a write access. Therefore, a gain in throughput can be achieved by delaying the supply of the write data relatively to the write instruction on the control bus by the same amount of time as the SRAM delays the read out of data relatively to the read instruction. This mode of operation is called *late write*. SRAMs supporting this mode of operation are called *Zero Bus Turnaround (ZBT) SRAMs* or *no-turnaround SRAMs*. Finally, there are *double data rate (DDR) SRAMs* which use both edges of the clock to transfer data.

4.2 The role of the memory controller

Fig. 41 shows a minimal computing system with a central processing unit (CPU), a memory controller, and memory chips forming the main memory. The memory controller is responsible for the translation of read and write requests of the CPU into control signals for the DRAM array. In addition, the controller must satisfy the timing requirements of the memory chip because the timing of control signals is not checked by the DRAM itself. The memory controller may stall the issue of new requests by the CPU until the DRAM is again capable of accepting read or write accesses. As we have learned from the preceding description of RAM properties, the delay penalty caused by activation, precharging, and bus turn-arounds does not only depend on the type of access but also on the inner state of the RAM and the order of the accesses. An optimized controller should therefore keep track of these aspects.

The format of the memory accesses issued by the CPU depends on the CPU architecture and is usually set at start-up time. The size of an access equals the size of a cache line so that a single burst transfer is sufficient to exchange the contents of a cache line. Embedded CPUs often use a memory controller integrated into the CPU core. Personal computing systems rather use an external controller through a front-side bus for more flexibility in the system design accepting an additional delay of one to two bus clock cycles per access.

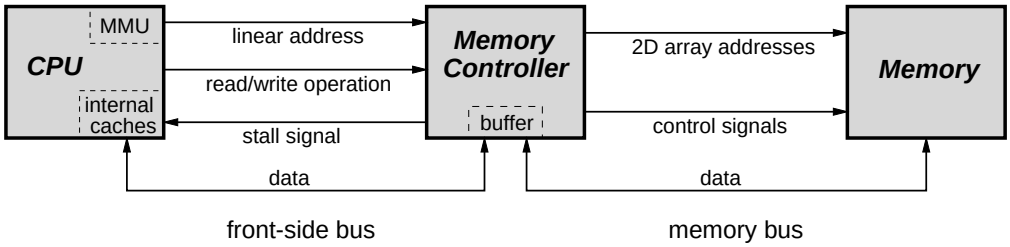

Fig. 41: A minimal computing system.

Memory controllers without additional operation queues are not able to reschedule read and write operation requests from the CPU. Most of the memory controllers currently found in PCs and embedded systems belong to this class. However, the latency of a memory operation can be reduced by using heuristics which exploit the row-wise organization of DRAMs. For instance, leaving memory rows activated as long as possible to save some precharge and activate operations is called an *open-page* policy. In contrast to that, a *closed-page* policy precharges an active memory row as soon as possible. This method will be employed if row or bank changes appear frequently. Moreover, the controller may buffer additional memory requests of the CPU. Such a controller is able to exploit the pipelined memory interface of modern DRAMs by overlapped processing of the current and the next pending accesses. Requests cannot be rescheduled but the sequence of control signals can be shortened. Finally, the memory controller may implement simple schemes to map physical address ranges to other regions by address permutations for interleaving bank accesses.

Besides a full-custom design there are some other design choices to implement a memory controller:

- *Integrated memory controller:* Processors aimed at embedded systems such as network processing often already include one or several memory controllers. The implementation reduces to a configuration of some operation mode registers in this case. Integrated controllers have the advantage to facilitate the communication between the CPU core and the controller without the overhead of using a front-side bus.

- *Stand-alone controller:* A variety of stand-alone controllers is available which are used as supporting chip sets of CPUs. This type of controller usually allows more flexible RAM configurations and organizations at the expense of an additional front-side bus.

- *Synthesizable block:* Memory controllers are also available as pre-designed building blocks for a given technology or in a high level design language such as VHDL. The controller can hence be adapted to individual requirements.

4.3 Performance model of a memory subsystem

This section describes a performance model of a memory subsystem consisting of a CPU, a memory controller, and RAM. The model is represented by a Petri net [115, 128], a formalism with a visual notation. The model is developed with the goal to have a graphic documentation and to analyze the interplay of the different components. Animated simulation moreover ease debugging so that deadlock and race conditions can fast be found. We especially focus on the functionality of the memory controller since its architecture and the inner state are not well documented by manufactures. Its required functionality is basically reverse-engineered from descriptions of DRAM data sheets. Additional features are added afterwards to exploit inherent parallelism of the inner DRAM architecture. We therefore begin with a description of the memory model and continue with the memory controller whose properties heavily rely on the memory features. The performance models derived in this section will serve as a basis for the performance design-space exploration in the next section.

4.3.1 Why use high-level Petri nets?

State machine based approaches, such as StateCharts [79] or ROOMchart [139], are best suited for control dominated systems and suffer from their inability to express data flow. Another discrepancy between a system and its model representation can be found looking at all the tools that do not allow to express structural similarity between a system and its model, e.g. spreadsheet based models and models written exclusively in the form of programming language code.

The use of object-oriented modeling [19, 136] becomes more and more common. Although object-oriented formalisms contain several features to produce detailed models, they are not intended to be executable as such. Place/Transition Petri nets [115] have several desirable properties, such as being intuitive, graphical, able to express concurrency and data flow. However, they are confined to the use in small scale models because a concept of hierarchy is missing. High-level Petri nets (such as Coloured Petri Nets [91]) are best suited, since they have an expressive inscription language and also structuring features.[1]

Since the main focus of the investigations is on performance issues as the determination of the data bus usage and throughput as well as the duration of schedules, the option to associate periods with transitions in timed Petri Nets is preferred over the option to formally check properties like liveness or reachability of a Place/Transition net. Furthermore, it is an advantage to use colored tokens and guard functions because the size of the whole net becomes sufficiently small by shifting some complexity into transitions and tokens to facilitate a quick overview of the entire system. We thus employ a similar methodology as the one presented in [25] in which by modeling the instruction execution of a microprocessor the same problems are dealt with as, for instance, data-dependent conditions of action execution and accurate instruction flow representation. Finally, the similarity between the hardware system and its model is additionally supported by (hierarchically) using subnets as components.

4.3.2 Modeling environment

The CodeSign tool is used to model and simulate a memory subsystem represented by a Petri net [115, 128]. A detailed description of the tool can be found in [46]. CodeSign is freely available [4]. It is based on a kind of colored Petri net that allows efficient modeling of control and data flow. CodeSign's specific properties can be summarized as follows:

- *Components, composition, and hierarchy:* Components are subnets of Petri nets with input and output interfaces that are applied to interconnect components. Inside components, input interfaces are connected to places and transitions are connected to output interfaces. Linking output with input interfaces, components are directly interconnected maintaining Petri net semantics. In Fig. 42 the model of a RAM basic cell with its interfaces is shown as an example. The model is explained in detail in section 4.3.4. If input interfaces are connected together, a single token will produce several tokens with the same data value for each component. Connecting output interfaces together is equivalent to connecting several transitions to a single place. Therefore, the examples in Fig. 43 are equivalent. Thus, models can be hierarchically structured and components can be reused. Moreover, as interfaces do not disturb Petri net semantics, a flat Petri net with the same functionality can always be generated from the hierarchically structured net.

[1] An introduction to Petri nets can be found in Appendix A.

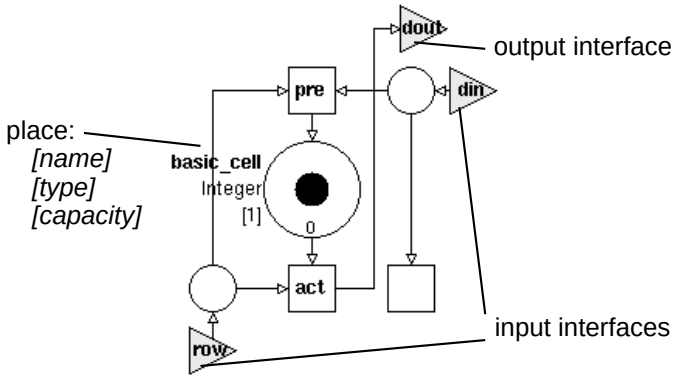

Fig. 42: Petri net model of a RAM cell component.

- *Object oriented concepts:* Components are instances of classes that are arranged within an inheritance tree. That is, classes inherit features and (token) data types from their parents. They may contain functions which can be used in transitions. Functions are written in an appropriate imperative language. With these facilities, incremental updates and evolution of models and components are supported as well as configurability and parameterization.

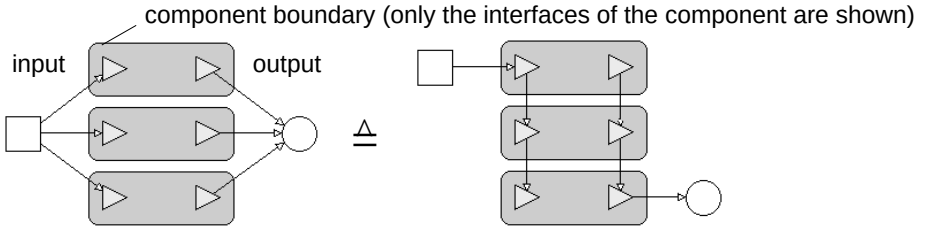

Fig. 43: Using interfaces in CodeSign.

- *Notion of time and simulation properties:* An enabling delay interval can be associated with each transition. As a consequence, tokens carry time stamps containing information about their creation date. Models and components can be inspected and simulated at all levels of abstraction. That is, performance evaluations and functionality checks like race conditions and timeouts can easily be performed. Finally, the simulation can be animated at run-time.

4.3.3 Overview of the model

In this subsection, the modeled memory subsystem is described which includes a memory controller, a synchronous memory chip as well as the relevant parts of the CPU and the data bus. In Fig. 44 an overview of the system is given. The CPU issues *data read* and *data write* requests. Each request token carries information about the type of the request (read or write) and the address where

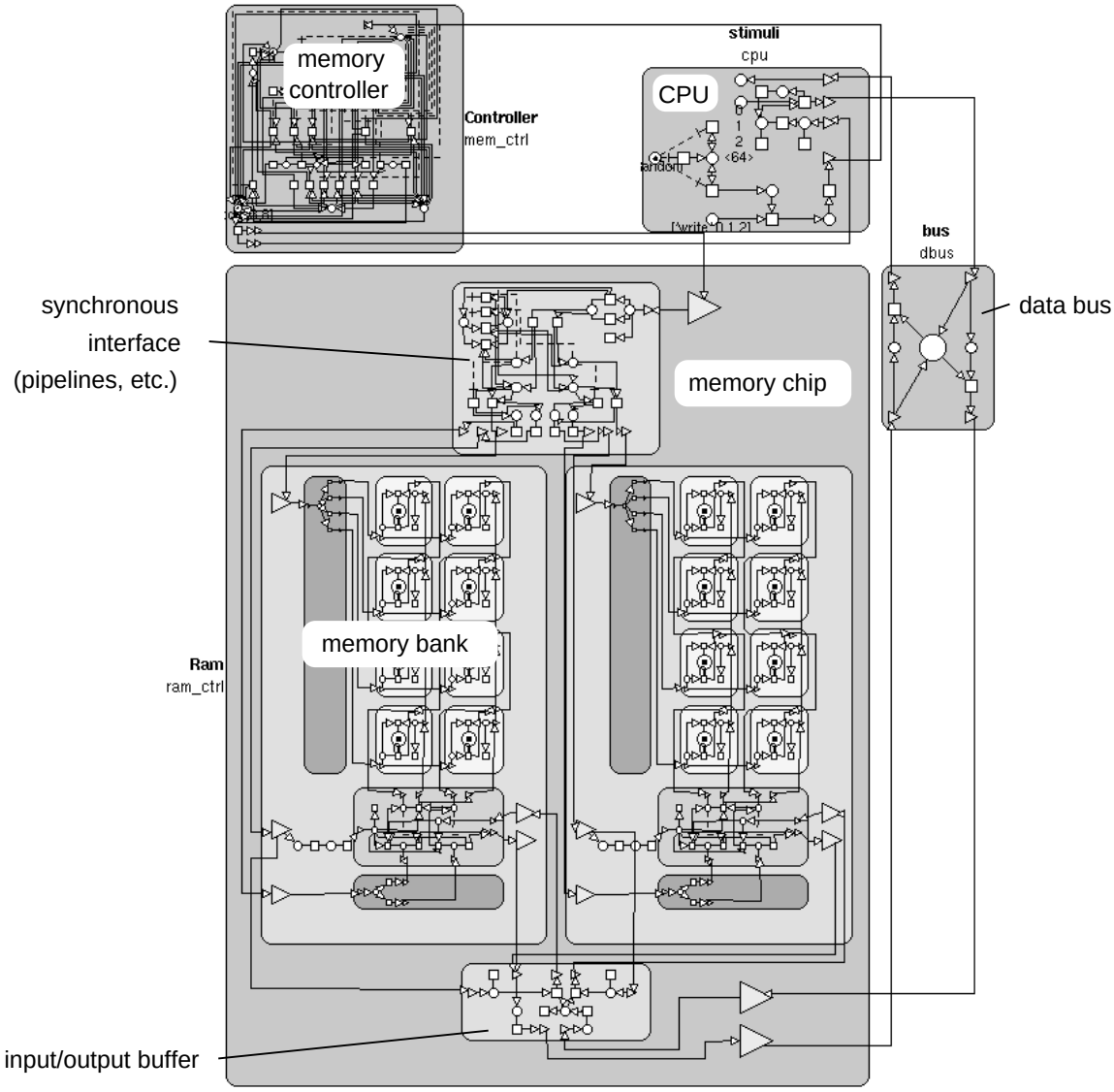

Fig. 44: CodeSign screen shot showing the model of a memory subsystem.

the data can be found in memory. The sequence of requests can arbitrarily be chosen and may be extracted from address traces generated by a tool like [35].

After having received a request token from the CPU, the memory controller preprocesses the token for the SDRAM chip. The address value of the token has to be split into memory bank, row, and column addresses as these values must be transferred at different times to the RAM chip. Besides, additional instruction tokens are generated which are necessary to prepare the memory, in particular tokens that initiate activate or precharge behavior of the memory cells. The memory controller is explained in more detail in section 4.3.5. At this stage, the instruction tokens may be seen as micro instructions for the control logic of the memory chip which have been created from the relatively coarse grained read and write requests of the CPU.

Finally, the memory chip obtains the modified tokens from the memory controller. An appropriate memory data transfer is initiated. The model of the memory chip is described in the next section. At last, the requested data item is put on the data bus and either received by the controller in case of a read request or received by the RAM in case of a write request.

The explanation of the model is organized "bottom-up" beginning with a description of the memory chip itself and ending with the subsystem overview. The description of the memory controller is the most precise one since our main goal is to reveal its properties.

4.3.4 SDRAM architecture

Asynchronous RAM array: In Fig. 45, a conventional memory bank with its sense amplifiers, row, and column decoders can be seen. The decoders are realized using mutual exclusive guard functions within the transitions because only a single transition may be enabled per address. The data within a memory cell is represented by a token of the type integer in the center place as it can be seen in Fig. 42. When a row of the memory array is selected by an *activate* instruction issued by the memory controller – a token containing a row address value, a memory bank address, and an operation identifier –, all data tokens in that row of memory cells are transferred to the sense amplifiers.

Sense amplifiers: After the activation of a memory row, the data tokens in the sense amplifiers can be read or updated column by column. The read or write state is reached when the memory controller issues *read* or *write* instruction tokens each containing an operation identifier and a column address. Read and write operations are destructive. Once the information is transferred from the cells to the sense amplifiers, the charge in the memory cells is lost. Therefore, the information in the sense amplifiers must be written back to the cells when the data of that special row is no longer needed. This operation is forced by a *precharge* command of the memory controller. In this case, the integer tokens carrying the memory information are written back to the cells.

Synchronous interface: The memory chip is completed by adding an input/output buffer pair, registers, control logic (the synchronous interface), and by using multiple memory banks that have to share buffers. The control logic implements several instruction pipelines, one for each memory bank. The pipelines are represented by successive transitions with an enabling delay of one clock cycle and places with limited capacity. That way, tokens carrying operations and addresses are delayed until the asynchronous memory array has finished the previous operation. Furthermore, the address may be incremented for burst transfers of consecutive addresses. Thus, memory banks are able to work concurrently due to separated instruction pipelines, but only a single one is actually able to transfer data through the shared input/output buffers at a time.

All timing parameters described in subsection 4.1.1.2 except the refresh period are considered in the CodeSign model. All modeled delays within an SDRAM component are derived from these timing constants.

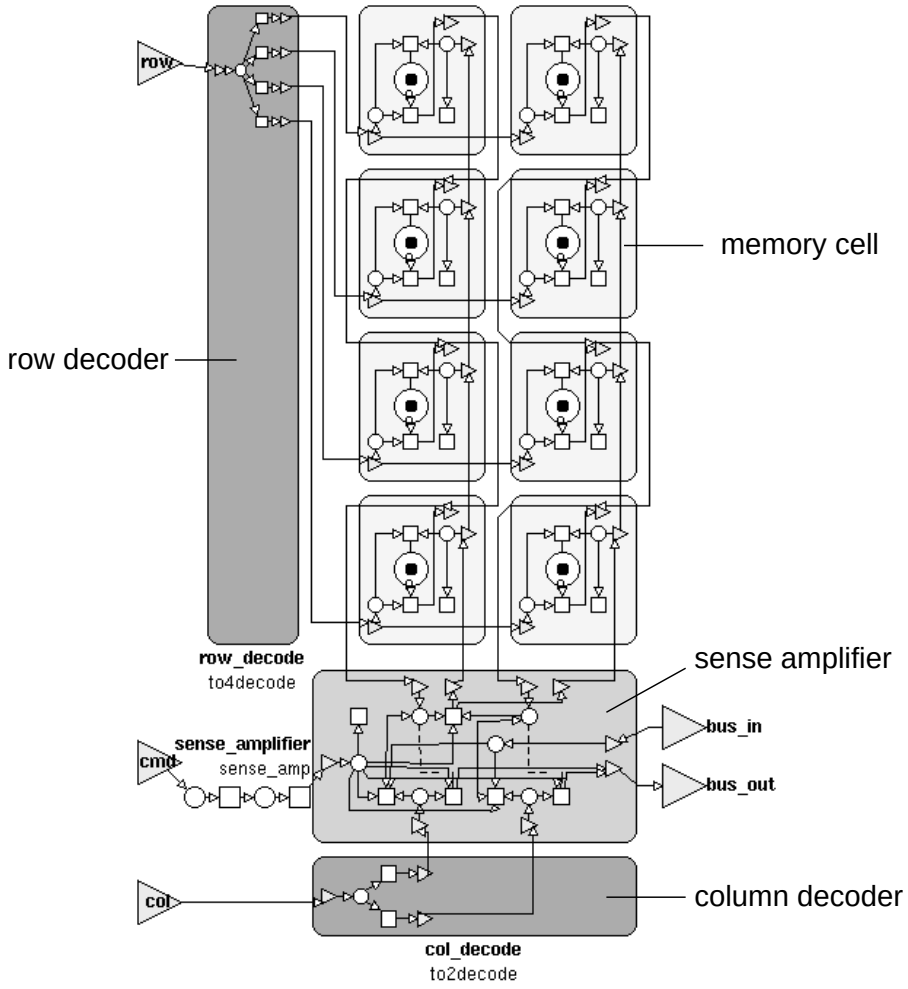

Fig. 45: A conventional asynchronous memory bank with $4 \times 2 = 8$ memory cells.

4.3.5 Memory controller

Functionality: A memory controller receives requests from the CPU at arbitrary times. They consist of the type of operation (read or write) and address information (typically 32 Bit addresses for a bytewise linearly addressable memory space). Sometimes, this address information has been preprocessed by a memory management unit (MMU, see Fig. 41) that usually maps addresses to other memory regions due to memory page limits or protected memory regions (e.g. reserved for memory mapped IO). The memory controller now has to take care of delays of the memory array. Since there typically are several memory banks within a memory chip, the controller must prevent data tokens from collision with other tokens, e.g. on the data bus. The memory chip will translate the instructions coming from the controller into actions on the memory cells without any timing checks.

Petri Net model: At first, the controller has to split the address information of the request tokens from the CPU into several tokens since memory chips are or-

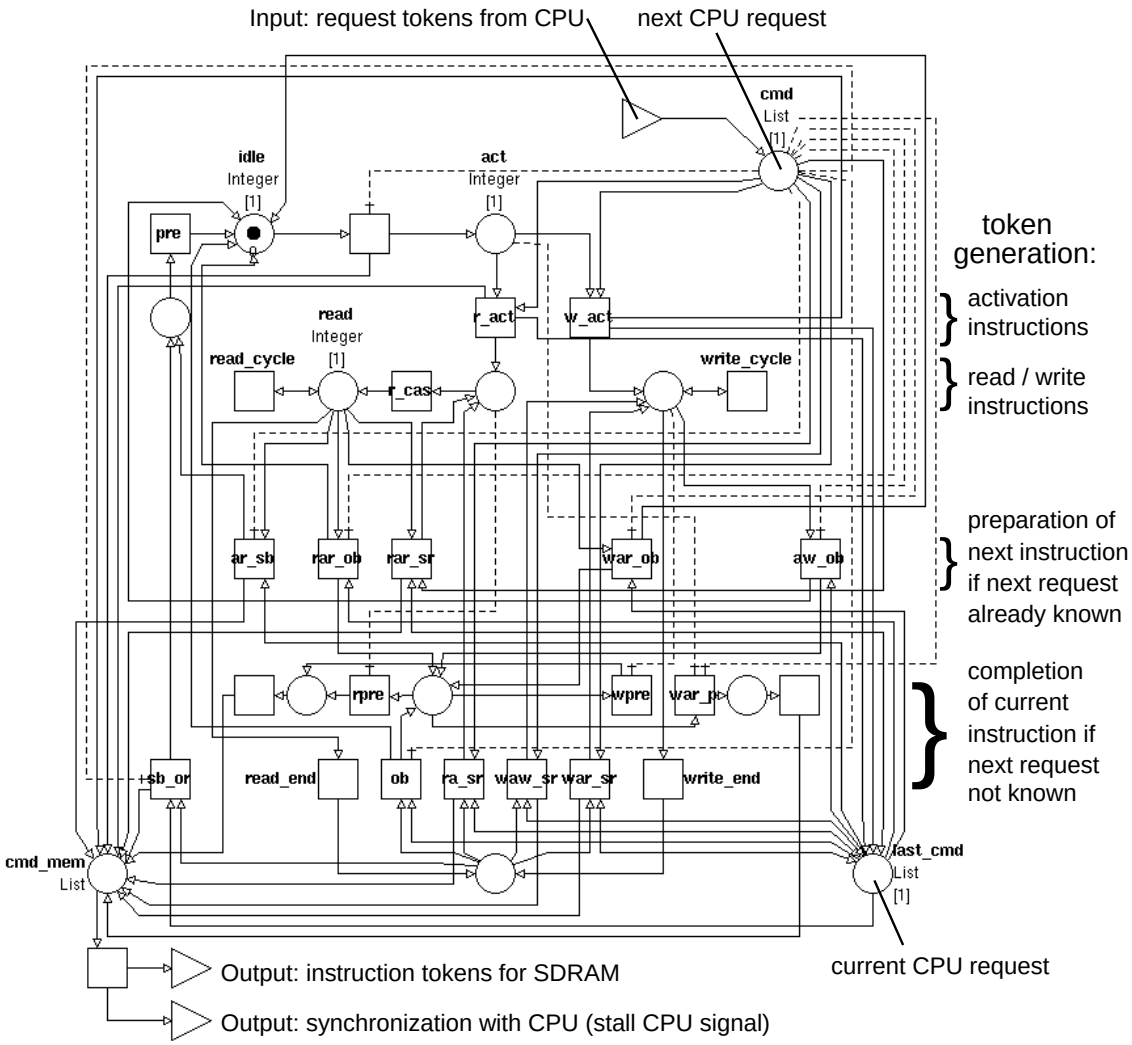

Fig. 46: Model of the memory controller.

ganized as two-dimensional arrays. The split address information is transferred to the selected memory chip at different times within the memory access cycle. The respective memory row must be opened with an *activate* command (token) which consists of an instruction identifier and bank and row addresses. Then, the request token itself is transmitted to the memory chip which consists of an instruction identifier of a read or a write operation and the bank and column addresses. Read and write operations on the same memory row can be repeated several times until finally the memory row must be closed. For this, the controller transfers a *precharge* instruction token holding an identifier for this type of operation as well as bank and row addresses. This control flow-dominant part of the system resembles the behavior of a finite-state machine. When the current state changes, actions are performed so that an instruction token according to the current state of the system is issued to the RAM.

In Fig. 46, the memory controller is shown. Dotted arrows are *read only*

connections. In principle, they could always be replaced by a read and a write connection because the token data is read from a place, never modified, and returned to the place. The controller can be seen as two distinct Mealy-like finite-state machines, one for processing read requests from the CPU and another one for processing write requests. In Fig. 47, the state machine responsible for processing the read requests is shown (extracted from Fig 46). Places that keep

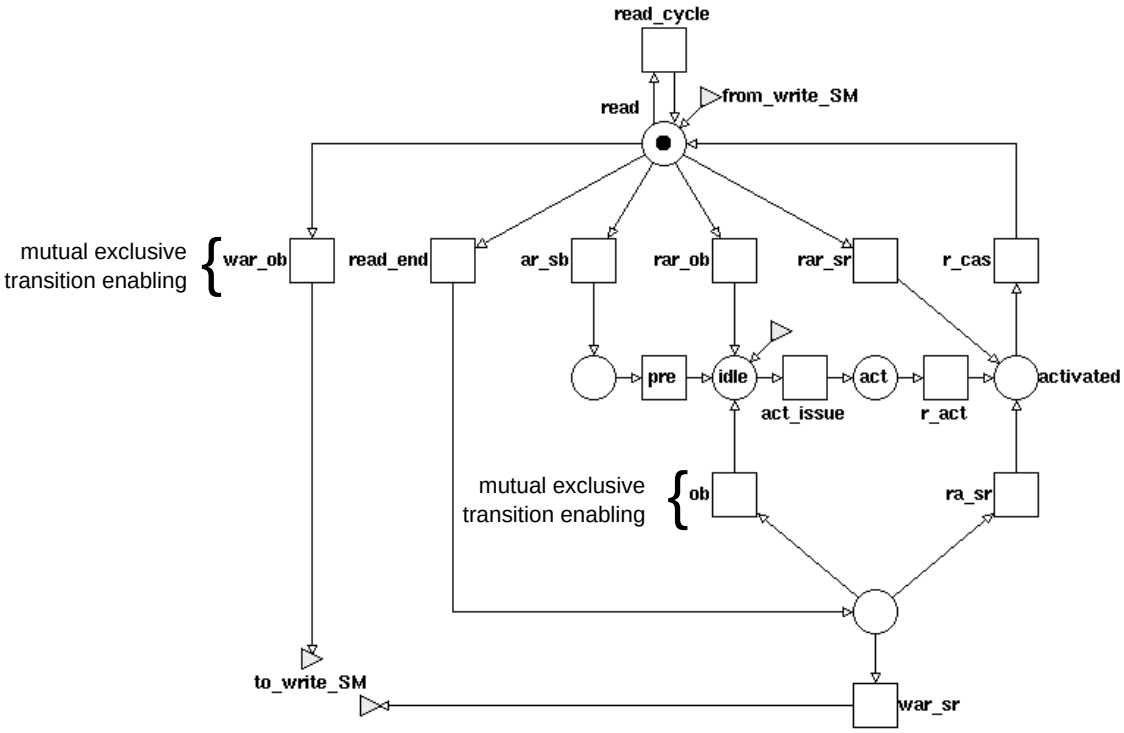

Fig. 47: State machine handling the read requests.

track of the global state – the request token currently in process – or that buffer request and instruction tokens at the interfaces of the controller are not shown. In other words, Fig. 47 only shows the state transitions and not the input and output places which are responsible for mutual exclusive transition enabling.

Ex. 10: **(Memory access cycle)** *Let us assume that a read request currently is in process, as shown in Fig. 47, and another read request which is already known in advance should be processed next. This next read wants to access another memory row of the same memory bank. Thus, the transition* ar_sb *is enabled (*ar_sb *stands for "read after read, same bank"). As an example of a guard function, the condition for enabling the transition* ar_sb *is shown in Fig. 48. The function checks the different value fields of the current tokens in the places* cmd, last_cmd, *and* read. *The places* cmd *and* last_cmd *are displayed in the top-right and the bottom-right corners of Fig. 46. The transition* ar_sb *fires after a certain amount of time which depends on the used SDRAM and the chosen burst transfer mode. This firing generates an instruction token for the SDRAM that starts*

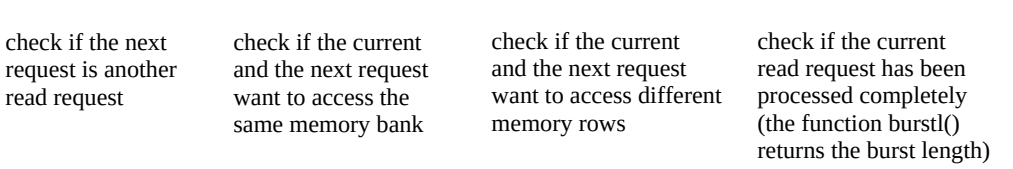

Fig. 48: Guard function of the transition ar_sb.

the precharge of the corresponding memory row. Then, the transition pre *is enabled and fires after the SDRAM precharge time* t_{RP}*. Therefore, when reaching the place* idle *the corresponding memory bank will be in the idle state and the controller can issue another instruction token for the SDRAM by firing the transition* act_issue*. This instruction token activates the corresponding memory row for the next read request. The transition* r_act *fires after the SDRAM activation delay* t_{RCD} *and generates a read instruction token for the SDRAM. Finally, the transition* r_cas *models the CAS latency* t_{CAS}.

In Fig 49, a black box view of the controller is given in which three arbitrary read request tokens from the CPU produce six instruction tokens on the output interface of the memory controller. All tokens for a whole memory cycle containing an activation, two reads, and a precharge can be seen followed by another activation and a read instruction. The time stamps in this example are integer values as requests and instructions are transferred at multiples of the clock cycle time. The flow of instructions from the controller to the RAM is

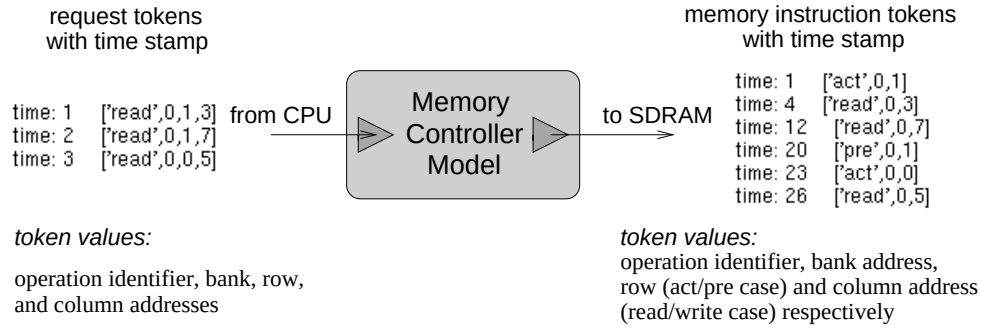

Fig. 49: Instruction token generation by the memory controller model.

modeled using tokens which consist of a string value and two integer values (see Fig. 49 on the right). This way, the tokens emulate commands with an operation field (read, write, precharge, activate) and two address fields (bank address, row and column address respectively). Besides, a token with an integer type is used within the controller to model the current state. The value is used as relative time reference in clock cycles during the whole memory cycle of the corresponding read or write request.

The read-after-read access explained in ex. 10 can be observed between the second and third request token in Fig 49. The read-after-read request for an

entry in the same memory bank but in a different row causes the generation of two additional instruction tokens for precharge and activation. Moreover, some tokens must be delayed to cope with the timing restrictions of the SDRAM.
Model characteristics: Basically, the controller behaves like commonly used controllers in PCs, i.e. it schedules read and write requests of the CPU sequentially without affecting their order. However, it is capable of abbreviating the latency penalty in case the next request by the CPU is already known by overlapping the processing of control signals for consecutive requests. The penalty is usually introduced due to a change of the memory row or bank. This feature will be described in more detail in Section 4.4.

4.3.6 Memory subsystem

The memory subsystem is completely modeled by adding a data bus and the I/O part of a CPU. The data bus is shared for read and write data and therefore realized by a mutual exclusive access scheme. Read and write request tokens may be extracted from previously generated address traces. The flow of read and write requests from the CPU to the memory controller is modeled using tokens which consist of a string value and three integer values (see Fig. 49 on the left). Thus, the tokens may be seen as instructions with an operation identifier (read or write request) and one big address field which is already grouped into three main areas (bank, row, and column address) e.g. by a memory management unit. In addition, the controller may stall the CPU in order to synchronize data transfers between CPU and RAM. Note that we model a memory controller which is supposed to be integrated into the CPU since the data bus directly interconnects RAM and CPU.

4.3.7 Conclusion for the performance model

With the help of the quickly developed performance model, promising performance increases have been deduced for particular access patterns. A more detailed description is published in [68, 65, 70]. Further investigations in the following section focus on examinations of real applications to quantify how often the different access patterns are actually used. Accordingly, virtual performance improvements will be determined which depend on the processed application and the properties of the memory and the controller used in the system.

4.4 Performance impact of controller and DRAM type

Based on the insight derived from the analysis of the Petri net-based performance model of a memory subsystem, we are now able to investigate the performance impact of memory controller operation modes and the interplay with different DRAM types for selected benchmarks. In order to simulate memory access patterns generated by whole applications running on a CPU, performance

models of a memory controller and DRAMs have been incorporated with a mature CPU simulation tool set. The Petri net model has especially contributed to an accurate implementation of the open-page operation mode and overlapped processing of the memory controller which are described later in this section. The impact of the underlying memory system on the performance of a computing system is explored by an extensive simulation of a variety of benchmarks, memory controller configurations, and DRAM types. More background information about the performed simulations can be found in [67, 71].

The section starts with a description of the features of the CPU simulation environment and the properties of the added memory controller and DRAM types. We define the set of applications used as benchmark and finish with an analysis of the simulation runs.

4.4.1 Simulation environment

SimpleScalar tool-set CPU simulator: We use SimpleScalar [23], version 3.0a, for cycle-true, execution-driven simulations. The architecture of the instruction set of the simulated CPU, called Portable ISA (PISA), has been inspired by the MIPS IV instruction set [125]. The CPU's internal data path is 32 Bit wide. There are separate reorder buffers for compute and load/store instructions and register renaming facilities to make out-of-order instruction execution possible. The five stages of the pipeline can be reconfigured as well as the number of execution units. The cache levels use writeback mode and non-blocking loads and stores.

Simulator enhancements: We have integrated the following DRAM and memory controller models into SimpleScalar. An SDRAM-based system represents a mature and widespread RAM configuration whereas the RDRAM configuration considers the recent approach to use narrow memory buses at high clock rates with an increased number of internal memory banks.

Tab. 15: SDRAM timing parameters.

t_{RCD}	20 ns	*row address strobe to column address strobe delay:* time to activate a row of an idle memory bank
t_{CAS}	20 ns	*column address strobe delay:* time between issuing a read instruction and the appearance of the first data item
t_{RP}	20 ns	*precharge time:* time to precharge an active row
t_{DPL}	20 ns	*data input to precharge delay:* time needed from the last data bus cycle of a burst write to a following precharge of the bank
t_{WAR}	20 ns	*write after read delay:* delay being subject to bus turn around limitations of the DIMM

- ***SDRAM:*** A mature PC100 [88] compliant SDRAM of the newest 256 MBit generation with a 16 Bit organization, four internal memory banks, and a row size of 1 KByte is modeled. Hence, four devices have to be controlled in parallel to form a common 64 Bit wide memory bus, similar to a DIMM with four mounted devices. The controller sees memory banks four times the size of a single chip's bank. The memory bus is clocked at 100 MHz. The following timing parameters described in [88] are considered, see Tab. 15.

  The CPU uses 32 Byte cache lines which corresponds to burst transfers of length four. Refresh operations take less than 1% of the run-time and are neglected.

- ***RDRAM:*** The RDRAM memory bus is 16 Bit wide as is an RDRAM chip. Again, 256 MBit devices are modeled. The Rambus channel consists of four RDRAMs. The channel configuration allows to control the banks of each memory chip individually. The chosen RDRAM [127] consists of 32 internal memory banks with a row size of 2 KByte. However, adjacent banks have to share sense amplifiers. The Rambus channel runs at 400 MHz using both edges. The following timing parameters described in [127] are modeled, see Tab. 16.

Tab. 16: RDRAM timing parameters.

t_{RCD}	17.5 ns	*row to column address strobe delay*
t_{CAC}	20 ns	*column address strobe delay*
t_{CWD}	15 ns	*column address strobe write delay:* time needed from the end of the control packet which initiated the write access to the beginning of the corresponding data packet
t_{RP}	20 ns	*row precharge time*
t_{RTR}	20 ns	*retire delay:* time needed for data to be transferred from the write buffer of the device to the corresponding sense amplifiers
t_{RDP}	10 ns	*read to precharge delay:* time needed from the end of a control packet initiating a read access to the end of a precharge control packet
t_{PP}	20 ns	*precharge to precharge delay:* for precharge control packets in the same device
t_{packet}	10 ns	time to transmit a packet

  The CPU's cache line size corresponds to two data packets transferred over the Rambus channel. Again, refresh operations which take 1.0% of the run-time have been neglected.

- ***Address translation:*** The memory controller has the option to map the physical addresses to chip, bank, row, and column addresses of the main memory. Two variants are implemented.

  - *linear translation:* The memory controller maps physical addresses in a linear manner to the chips and banks of the main memory. That is, passing the boundaries of a DRAM row results in an access of the next row in the same internal DRAM bank.
  - *interleaved translation:* Memory rows are mapped in an interleaved manner to the internal banks of the DRAMs. Our generic address permutation scheme results from the row size of the DRAM. One may think of interleaving divisions of a row. These address permutations however must individually be derived from the software and data sets used. In our scheme, whenever the address passes the boundaries of a DRAM row, the next address accesses another bank and not the next row of the same bank. This mode will be beneficial if an application accesses streamed data since changing to another memory bank is usually faster than accessing another memory row in the same memory bank. Due to shared sense amplifiers every second bank is allocated consecutively for streamed data in the RDRAM case.

- ***Activation policies:***
  - *strict open-page policy:* Memory rows are kept activated as long as possible by the controller. This way, accesses bounded to the current row do not need to precharge and activate the row again. However, the controller needs additional open-page counters to keep track of activated rows.
  - *closed-page policy:* The controller precharges an active row as soon as possible. If row changes happen frequently, this mode will result in better performance because some of the precharge time can be hidden.

- ***Overlapped processing:*** The controller is able to process an access on the memory bus while receiving further requests from the CPU. The CPU must support this mode of operation by non-blocking load/store execution. The controller may buffer requests to hide some of the latency introduced by activation and precharge tasks, taking advantage of the pipelined interface of recent DRAMs.

Ex. 11: **(Overlapped processing of memory accesses)** *An example is given in Fig. 50. The second operation is a read access to another bank of the memory than the bank of the preceding write access. Thus, a row miss occurs leading to additional precharge and activation commands. Now assume that the second operation, the read access, is known during the processing of the first access, since the corresponding request has already been transferred from the CPU to the controller. The latency for precharging and activation can completely be hidden in this example.*

The amount of clock cycles which can be overlapped depends on the current as well as the buffered accesses and the inter-arrival time of the requests at the memory controller. The requests are not rescheduled but stored in an in-order queue. Overlapped processing has not been modeled in closed-page mode.

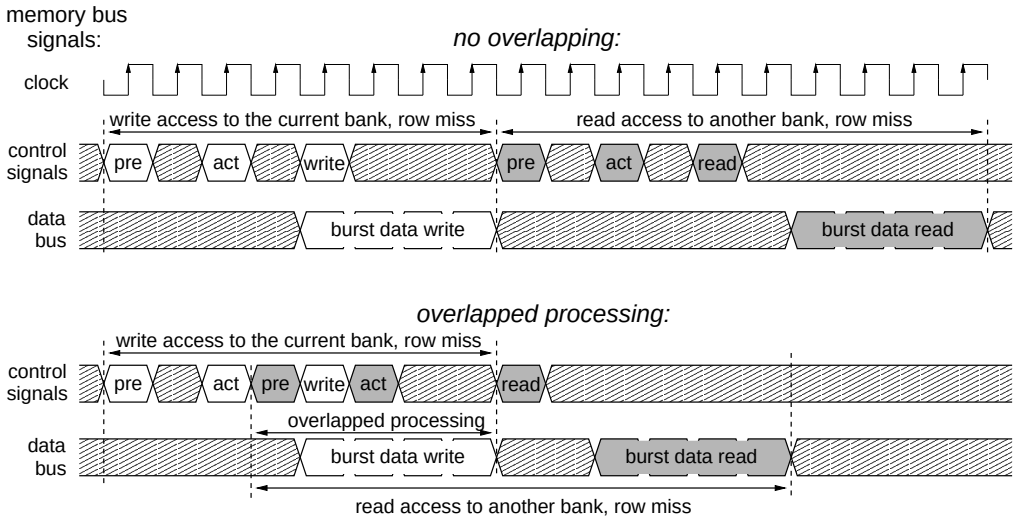

Fig. 50: SDRAM, overlapped processing.

Therefore, linear and interleaved address translation are not distinguished in closed-page mode since both translation schemes result in the same latencies.

In closed-page mode, two different cases must be distinguished for SDRAMs and RDRAMs because only the request type (read or write) is of importance. Using open-page mode and SDRAMs without overlapped processing, 18 different cases are determined depending on the request type of the current and the buffered operations, the bank addresses, and the row state (row hit/miss). For RDRAMs, 36 main cases have been identified. The higher number of cases compared to the SDRAM controller is mainly caused by shared sense amplifiers, write buffers, and the use of several concurrent devices. If the overlapped processing mode is enabled in addition, the overlapping period will be of variable size. The period of time a request occupies the memory is determined at the arrival time of the request at the controller.

4.4.2 Experimental results

4.4.2.1 Applications

A mixed collection of programs is chosen to generate memory access patterns with high diversification, see Tab. 17. Sorting and searching tasks are a mandatory part of every network processor to lookup routes, sort packets according to dynamic priorities, manage queues, etc. All kinds of data compression and media processing will be employed if a networking node is the source and/or the destination of multi-media traffic. This is very likely the case for a networking node situated at an access link to the Internet. Finally, the program compilation is augmented by selected mature CPU benchmarks and some other applications. In this way, the set of programs chosen for evaluation of a memory subsystem shows greatly varying features. Tab. 18 lists selected properties of some characteristic programs. The first two columns present cache miss rates for instructions

and data. The last two columns reveal the relative amount of executed load/store and branch instructions for a typical simulated program run. One can see that the share of cache misses and executed instructions exceptionally deviates from program to program. Therefore, any potential and future network processing task should closely match the characteristics of one of the chosen applications for evaluation so that the results of this section can potentially be applied to a variety of programs.

Tab. 17: Programs for simulation. Those programs which are freely available have been compiled from different sources at [12].

CPU benchmarks	
spec95 ijpeg	Image compression (input file *specmun.ppm*), integer performance rating.
spec95 li	Lisp interpreter (input *test.lsp*), integer performance rating.
spec95 swim	Finite difference approximations (input file *swim.in*), floating-point performance rating.
linpack	Linear algebra routines, floating-point performance rating.
Voice / video / data compression	
g723	ITU-T rec. g723.1 speech decoder integer reference code. Decodes test vector *dtx63.rco* encoded at 6.3 KBit/s.
g723enc	Corresponding voice encoder. The test vector *dtx53mix.tin* is encoded at 5.3 KBit/s.
tmn 3.1.2	ITU-T rec. H.263 video decoder by the Univ. of British Columbia and Telenor Research. 100 frames of a sequence encoded at 1 KBit/frame in qcif picture format are decoded.
tmnenc 3.1.2	Corresponding H.263 video encoder. 15 frames of a sequence are encoded at 3 KBit per frame.
gzip 1.2.4a	Compresses its own distribution archive (2.5 MByte) using Lempel-Ziv coding.
tfft	Fast Fourier Transform of 16 to 2^{16} samples (floating-point).
Sorting and searching	
c4 1.0	Connect-four game solver. Performs search operations on hash tables with more than one million 32 Bit hash entries.
heapsort 1.0	Sorts random heap-organized arrays with up to 2^{19} integers.
Other applications	
nsieve 1.2b	Computes prime numbers based on the algorithm of Eratosthenes. Integer arrays of several MBytes are used.
sim	Finds seven best non-intersecting alignments between two strings with 2500 characters using dynamic programming.

Tab. 18: Selected program execution statistics.

program	*cache miss*		*exec. load/store*	*exec. branch*
	instr. [%]	*data* [%]	*rel.* [%]	*rel.* [%]
swim	1.3	41.4	29.4	6.4
g723	0.5	0.1	13.4	22.3
tmn	1.2	8.5	55.5	10.5
c4	10.9	3.4	32.6	8.8
nsieve	0.0	25.9	11.7	28.1

4.4.2.2 Simulations

Settings: The following settings for SimpleScalar have been chosen to compare SDRAMs with RDRAMs under different memory controller configurations in an embedded system scenario. The CPU runs at a clock speed of 200 MHz and is two-way superscalar with out-of-order issue. Bimodal prediction with 512 entries is used. The reorder buffers have eight entries for compute and four entries for load/store instructions. The direct mapped first level cache is split into an 8 KByte instruction cache and an 8 KByte data cache. The cache line size is 32 Byte. There is no second level cache. Loads and stores are non-blocking. The following functional units are available: an integer ALU, an integer multiplier/divider, a floating-point ALU, and a floating-point multiplier/divider.

As described in subsection 4.4.1, the memory controller can apply linear (*lin*) as well as interleaved (*int*) address translation. Moreover, a closed-page (*c-p*), an open-page (*o-p*), and an open-page activation policy with overlapped processing can be chosen. We have not bounded the number of buffered requests in the performance model of our memory controller. However, the characteristics of the CPU architecture, especially the length of the reorder buffers, limit the number of pending requests. Finally, SDRAMs and RDRAMs can be controlled. The controller is supposed to be integrated in the embedded CPU.

Results: The simulated execution times of our set of applications are displayed in Tab. 19. The values are scaled to the corresponding execution time using SDRAMs and closed-page mode. Tab. 20 provides information about program execution statistics: cache miss rates, the amount of main memory accesses scaled to the number of all executed instructions (*total*), the ratio between the number of memory accesses that can make use of overlapped processing to the number of all main memory accesses (*overl.*), the amount of loads and stores scaled to the number of executed instructions (*rel.*), and the absolute count of executed load and store instructions (*abs.*). Since the amount of memory requests which can make use of overlapped processing does not vary remarkably by changing the address translation mode or the DRAM type, only the maximum of these simulation runs is considered in the table. Finally, Tab. 21 shows hit and miss statistics for the main memory system using open-page mode. Looking at two consecutive memory requests, the next pending request may access

Tab. 19: Simulated program execution times for different memory controller operation modes. The results are scaled to the execution time with an SDRAM-based system using closed-page mode.

program	*addr. transl.*	*SDRAM*			*RDRAM*		
		c-p	*o-p*	*overl.*	*c-p*	*o-p*	*overl.*
heapsort	lin	1.0	1.02	1.02	0.99	1.01	1.01
	int		1.01	1.01		0.96	0.95
tmnenc	lin	1.0	1.00	0.99	0.99	0.99	0.98
	int		0.99	0.99		0.97	0.96
g723	lin	1.0	0.99	0.99	0.99	0.98	0.98
	int		0.98	0.98		0.97	0.97
g723enc	lin	1.0	0.98	0.98	0.98	0.97	0.96
	int		0.97	0.97		0.95	0.95
jpeg	lin	1.0	0.99	0.98	0.98	0.96	0.94
	int		0.98	0.97		0.93	0.91
li	lin	1.0	0.97	0.95	0.96	0.93	0.88
	int		0.95	0.93		0.85	0.80
c4	lin	1.0	0.86	0.84	0.92	0.81	0.75
	int		0.86	0.84		0.78	0.72
swim	lin	1.0	1.15	1.06	0.95	0.84	0.69
	int		1.01	0.88		0.75	0.61
sim	lin	1.0	1.09	1.05	0.98	1.00	0.93
	int		0.98	0.93		0.77	0.70
nsieve	lin	1.0	1.10	1.03	0.96	1.05	0.93
	int		0.96	0.91		0.82	0.71
tmn	lin	1.0	1.06	1.02	0.95	1.00	0.93
	int		0.89	0.87		0.81	0.74
gzip	lin	1.0	1.09	1.05	0.96	1.04	0.99
	int		1.01	0.96		0.86	0.80
linpack	lin	1.0	1.06	1.06	0.96	1.02	1.01
	int		0.92	0.91		0.84	0.82
tfft	lin	1.0	1.03	1.01	0.96	0.98	0.94
	int		1.00	0.98		0.91	0.87

the same row in the DRAM as the current access. This situation is called a hit in the current bank (*cb hit*). If the pending request wants to access a different row in the same bank, a miss in the current bank occurs (*cb miss*). Finally, the situations where the pending request will hit or miss a row in another bank than the current one are counted as accesses to another bank (*ob*). The negligible amount of accesses to idle banks is not quoted in the table.

Analysis: For four programs the execution time is almost independent of the

Tab. 20: Program execution statistics.

program	*cache miss* [%]		*DRAM accesses* [%]		*exec. load/store*	
	instr.	*data*	*total*	*overl.*	*rel.* [%]	*abs.* [10^6]
heapsort	0.0	2.7	1.1	0.3	22.1	310
tmnenc	0.2	1.5	0.7	12.7	28.5	368
g723	0.5	0.1	0.6	3.0	13.4	120
g723enc	0.8	1.3	1.0	2.2	12.4	189
jpeg	0.7	3.7	1.9	17.4	25.2	147
li	4.5	2.3	6.4	9.3	47.9	495
c4	10.9	3.4	13.8	8.4	32.6	1229
swim	1.3	41.4	16.1	42.6	29.4	237
sim	2.3	17.9	11.4	18.2	36.3	1595
nsieve	0.0	25.9	5.9	45.0	11.7	113
tmn	1.2	8.5	7.1	27.3	55.5	250
gzip	0.0	17.0	5.3	28.6	28.0	138
linpack	6.1	8.9	4.5	3.7	26.9	21
tfft	1.0	7.7	5.5	15.7	38.9	80

memory and controller used: *heapsort*, the video encoder *tmnenc*, and the voice processing programs *g723* and *g723enc*. The programs show almost no misses in the cache. Although load and store instructions take around 20% of the executed instructions, only slightly more than 1% of the executed instructions result in a main memory access. This rate is too low to produce any run-time effects. For instance, looking at RDRAMs and the *tmnenc* program, the rate of expensive row misses can be reduced from 42% for linear address translation to 2% using interleaved address translation. However, there is no noticeable difference in the execution time.

For another group of programs – *jpeg*, *li*, and *c4* – using interleaved address translation always shortens the execution time. In most configurations however, the differences are negligible. The programs run faster by controlling RDRAMs instead of SDRAMs. Applying overlapped processing only improves the performance of RDRAM configurations noticeably. Using open-page mode instead of closed-page mode improves the execution time for both DRAM types. The programs have in common that they generate moderate miss rates in the instruction cache and almost no misses in the data cache, see Tab. 20. Since the share of loads and stores of all executed instructions is high (up to 48% for *li*), a noticeable amount of the simulated instructions results in main memory accesses. Moreover, the accesses to main memory are very localized. At least 56% of the memory accesses hit an already activated memory row. Thus, the memory controller well benefits from the open-page policy. The remaining amount of access misses is too small to influence system performance by changing from linear to interleaved memory translation.

Tab. 21: Main memory access statistics in percent of all memory accesses.

program	*addr. transl.*	*SDRAM [%]*				*RDRAM [%]*			
		cb[a]		*ob*[b]		*cb*[a]		*ob*[b]	
		hit	*miss*	*hit*	*miss*	*hit*	*miss*	*hit*	*miss*
heapsort	lin	4.4	91.7	2.7	1.2	3.1	59.5	6.8	30.6
	int		32.2	20.5	42.9		0.8	86.4	9.7
tmnenc	lin	30.9	45.6	15.6	7.9	30.1	31.5	28.0	10.3
	int		20.9	36.2	12.0		0.5	67.8	1.6
g723	lin	71.1	25.5	2.6	0.8	69.9	26.7	2.4	1.0
	int		6.6	19.7	2.6		0.0	29.8	0.3
g723enc	lin	57.7	26.1	14.6	1.6	57.3	26.5	14.6	1.5
	int		3.7	36.9	1.7		0.0	42.6	0.0
jpeg	lin	41.0	36.2	15.5	7.3	39.2	24.3	25.5	11.0
	int		19.5	26.6	12.9		0.7	58.7	1.3
li	lin	55.5	26.8	8.9	8.8	55.0	27.3	8.2	9.4
	int		13.3	17.0	14.2		0.3	43.3	1.4
c4	lin	79.5	8.9	8.0	3.6	77.3	9.5	8.0	5.2
	int		7.5	5.5	7.5		1.3	17.0	4.4
swim	lin	13.9	85.8	0.1	0.1	9.8	10.0	62.3	17.8
	int		11.1	34.8	40.1		0.0	88.9	1.2
sim	lin	30.1	59.1	4.9	5.9	30.1	59.1	4.9	5.9
	int		9.7	29.9	30.3		0.1	69.8	0.0
nsieve	lin	1.6	98.3	0.0	0.0	1.4	88.4	1.7	8.5
	int		24.9	54.4	19.1		0.3	89.3	9.0
tmn	lin	16.1	74.0	6.0	3.9	15.8	70.9	8.1	5.2
	int		9.8	65.4	8.7		0.2	82.5	1.5
gzip	lin	7.6	92.3	0.1	0.0	6.5	93.4	0.1	0.0
	int		25.5	32.7	34.2		0.1	92.5	0.9
linpack	lin	9.6	88.0	1.7	0.7	8.8	88.9	1.7	0.6
	int		15.2	67.4	7.8		0.1	91.0	0.1
tfft	lin	29.3	62.5	4.0	4.2	28.3	63.1	4.3	4.3
	int		42.0	19.8	8.9		21.7	33.7	16.2

[a] *cb hit/miss:* hit/miss in the current bank.
[b] *ob hit/miss:* hit/miss in another bank than the current.

The remaining group of programs, consisting of *swim*, *sim*, *nsieve*, *tmn*, *gzip*, *linpack*, and *tfft*, shows a noticeable performance improvement by using overlapped processing (except *linpack*). The execution time can be shortened up to 19% for RDRAMs using overlapped processing (*swim*, interleaved translation) and up to 13% for SDRAMs respectively (*swim*, interleaved translation) since up to 45.0% of all main memory accesses can make use of this mode.

Interleaved address translation shortens the execution time by up to 25% (*sim*, RDRAM, with overlapped processing). Since the programs generate high miss rates in the data cache combined with a high count of load and store instructions, at least 5.3% of all executed instructions result in a main memory access. RDRAMs benefit more from overlapped processing since their level of pipelining can better cope with bus turn-around situations. For instance at *nsieve*, for which almost 90% of the memory transfers have been classified as *read after write* or *write after read* accesses (not displayed), the RDRAM configuration shows a 13% increase in performance compared with only 6% for the SDRAM configuration. The significant performance improvements by using interleaved address translation instead of linear translation can be achieved because the programs show high DRAM miss rates in the current bank compared with the preceding group of programs. Using interleaved translation, these relatively expensive accesses can be exchanged for cheaper accesses in other memory banks. Moreover, RDRAMs benefit more from interleaving, since the high number of internal banks causes considerably less conflict misses. The high miss rate in the current bank also explains why the SDRAM configurations using open-page mode and linear translation perform poorer than the corresponding configurations using closed-page mode. A closed-page policy is able to hide some of the precharge overhead which may be caused by frequent memory row changes.

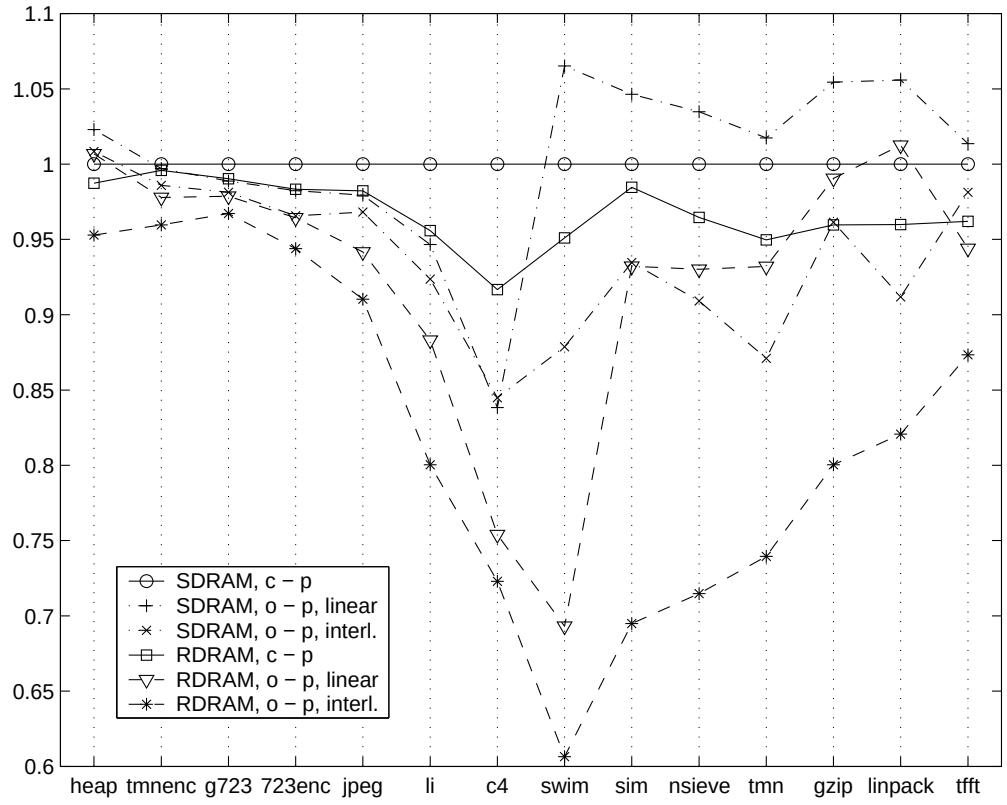

Fig. 51: Normalized simulated execution times for closed-page and open-page modes with overlapped processing. SDRAM with closed-page mode is used as reference.

Summary: The results are summarized in Fig. 51. On the one hand, the RDRAM configuration using open-page mode and interleaved address translation always outperforms all other configurations. The minimal performance gain ranges from some percent to up to 25% for the *sim* application. On the other hand, the RDRAM more heavily depends on the chosen memory controller access scheme due to the level of pipelining and the number of memory banks involved. In the worst-case, the *sim* program looses more than 50% of its best-case performance by using RDRAMs with closed-page mode. SDRAMs using open-page mode and interleaved address translation compete well with RDRAMs using linear translation. In the end, address translation influences the overall performance more than exploiting pipelined RAM interfaces.

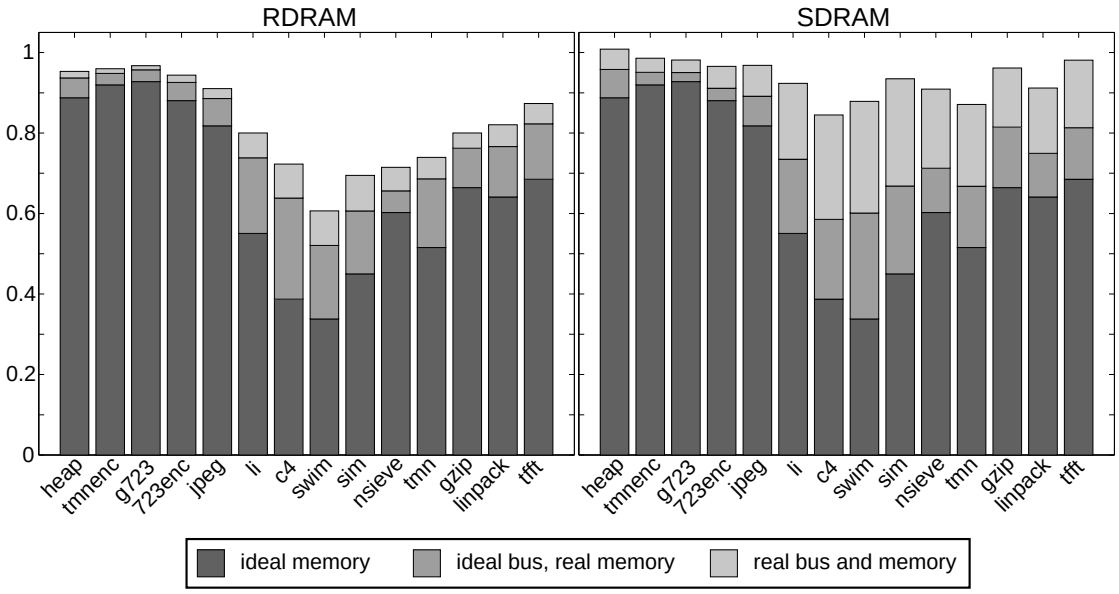

Fig. 52: Normalized simulated execution times for open-page mode with overlapped processing and interleaved address mapping. SDRAM with closed-page mode is used as reference. The influence of memory latency and restricted bus bandwidth is shown.

Additional simulation runs have been performed employing the methodology defined in [24] to separate the impact of raw bus bandwidth from the influence of DRAM latency. In this way, we are able to assess whether the Rambus performance advantage is based on architecture features or on raw throughput. A simulation run assumes an ideal memory with a single CPU-cycle delay. A second simulation run uses an ideal memory bus width but employs the DRAM latency models. A 32 Byte wide bus is assumed in the SDRAM case and a 32 Bit bus in the RDRAM case so that a cache line transfer needs a single data cycle and a single packet length respectively. The curves representing the fastest configurations – open-page mode with interleaved address mapping – from Fig. 51 are redrawn in Fig. 52. The execution time is divided into a share caused by inadequate bus throughput (upper part), memory timing (middle part), and raw computation time (lower part). One can clearly see that the performance advan-

tage of RDRAMs is mainly based on increased memory bus bandwidth. Only some of the benchmarks – swim, sim, nsieve, and gzip – can noticeably reduce the execution time by five to 10 % due to enhanced features of RDRAMs such as better bus turn-around behavior and a greater number of internal banks. Therefore, a further increase of the bus bandwidth is not a reasonable option but the properties of the memory core must improve for future memory solutions.

4.4.3 Conclusion for the design space exploration

The influence of recent DRAM technologies – SDRAMs and Direct Rambus RAMs – on embedded systems performance has been investigated. Especially, the impact of the memory controller has been explored by exhaustive simulation of a modern 32 Bit out-of-order superscalar CPU architecture together with different RAMs, applications, and memory access schemes. These schemes have been kept simple so that they can be implemented in custom-designed controllers, e.g. for network processors. The results from the simulation of 14 different applications on our modeled system can be summarized as follows:

- Looking at RDRAMs, applying an open-page access heuristic results in better system performance than a closed-page access scheme. SDRAMs however do not necessarily perform better in open-page mode.
- Exploiting the internal multi-bank structure of modern DRAMs by a row-wise interleaved address translation scheme accelerates the program execution by up to 25%. The amount of resolved capacity misses thus has always outperformed the newly caused conflict misses.
- Using additional in-order request buffers to overlap the processing of successive memory accesses may decrease the execution time by up to 19% for RDRAMs and by up to 13% for SDRAMs.
- The RDRAM system outperforms the SDRAM system by up to one third using the same memory controller settings. RDRAMs benefit from a higher bus throughput, more internal banks and a better bus turn-around behavior. However, RDRAMs depend more on the chosen access scheme than SDRAMs.

Thus, from a performance point of view, the transition from the mature SDRAM to the new RDRAM technology can be indeed beneficial for embedded systems. Besides, applying an interleaved address translation scheme has a greater impact on the system performance than exploiting pipelined memory interfaces.

4.5 Related work

Common memory controller features: Tab. 22 gives an overview of current SDRAM memory controllers. The support for a closed-page (*c-p*) activation

policy, for an open-page policy together with the maximal number of open pages (*o-p/#pages*), and for overlapped processing is shown.

Tab. 22: Memory controller features.

chip name	*c-p*	*o-p / #pages*	*overl.*
embedded CPUs with integrated memory controller			
Hitachi SH-4/7750	yes	yes / 4	yes
Intel 80960RN	no	yes / 8	no
Motorola Coldfire	yes	yes / 1	no
Motorola MPC8xx	yes	no	no
Motorola MPC8240	yes	yes / 4	no
NEC V832	yes	yes / 1	no
stand-alone memory controllers			
Intel 440BX	yes	yes / 32	yes
Motorola MPC106	yes	yes / 2	yes
NEC VRC5074	yes	yes / 8	yes
workstation class stand-alone memory controllers			
AMD 751	yes	yes / 24	yes
Digital 21174	yes	yes / 24	yes
Via Tech. KT133	yes	yes / 16	yes

Only few memory controllers are available for RDRAMs. Rambus offers two reference controller designs which can be used as a basis for own optimized ASICs. Address mapping and overlapped processing are supported and an open-page activation mode is mentioned as a future option without specifying the supported number of open pages. Intel's line of controllers (i820/i840/i850) has in common that only a small number of up to eight open pages can be used concurrently. This number is even less than the number of open pages supported by Intel's own SDRAM-based controller line.

Our modeled controller supports up to four open pages in the SDRAM system and 64 open pages for the RDRAM channel. In this way, we can be sure that the performance of our system is not bounded by a lack of open-page counters. We are hence able to evaluate the performance potential of using a high number of memory banks concurrently. Recent announcements of RDRAM controllers [77] even talk about the support of several hundred open pages. Our simulation model thus covers a reasonable set of memory controller features.

Issues in current research: Detailed cycle-true simulators of CPUs have been used for a variety of investigations to explore future processor systems together with their memory subsystem. These studies either focus on the influence of different CPU architecture features and cache designs on the memory behavior using standardized SPEC [146] CPU benchmarks [24, 49, 162] and particular

applications [135, 162] or concentrate more on the workload generated for the memory subsystem by certain applications [11]. These studies have in common that due to large second level caches the impact of the main memory is supposed to be low. The main memory is simulated by a simple model which only consists of two to three access delay parameters. Our study models the main memory system more precisely and emphasizes the role of the memory controller.

Enhancements of memory controllers such as stream buffers [110, 84, 124] and configurable complex address remapping functions [28] are rather complex and are thus unlikely to be introduced in embedded memory controllers in the near future. These concepts need adjustments in the compiler, may generate additional run-time overhead for reconfigurations, and may introduce additional delay for applications which do not show regular memory access patterns. We have restricted our study to simple memory controller architectures and enhancements which can already be found in current workstation systems.

Recent RAM surveys [126, 120, 97] focus on the functionality, the architecture, and typical applications of dynamic RAMs. However, they do not provide performance measurements under realistic workloads. An analysis of different DRAM architectures has been combined with performance investigations by simulation of a complex CPU model by Cuppu et al. in [41]. However, their objective differs from ours in that they simulate a workstation class computer with larger caches, longer cache lines, more superscalar units, and a higher clock frequency than embedded systems usually have. Cuppu et al. concentrate on an analysis and comparison of different DRAM types in order to clarify where time is spent during a main memory access. Opposed to that, we are interested in the impact of the memory controller on the performance of the whole computing system. We have restricted our studies to the two most recent DRAM types, but have investigated the influence of address permutations and pipelined accesses performed by the memory controller.

A performance study simulating forthcoming double data rate ESDRAMs and VC-SDRAMs combined with a 5 GHz, 8-way superscalar CPU is presented in [42]. The authors also focus on simple memory controller operation modes assuming that the circuit complexity of the controller will be rather limited at such clock speeds. Interestingly, they also come to the same conclusion for their high-end configuration as we do for embedded systems that address mapping has a greater impact on the performance than all other controller policies.

4.6 Summary

Since the main tasks of network processors such as packet header parsing and forwarding must not rely on caches, network processors only employ small caches (if at all). Accordingly, their performance particularly depends on the exploitation of RAMs. This is why this section has been devoted to the performance optimization of memory resources.

The functionality and timing constraints of synchronous RAMs have been explained to show why access delays not only depend on the type of operation but also on the inner state of the RAM and on the order of the accesses. Consequently, the role of the memory controller has been emphasized in order to make full use of RAM resources. A memory controller with enhanced features that take advantage of the inherent parallelism of multi-bank memory architectures with pipelined interfaces has been reverse-engineered and modeled by a visual formalism to ease the understanding of the interplay of controller and memory. Based on the gained insight performance models of different controllers and RAM types have been incorporated into a mature CPU simulator. The influence of recent DRAM technologies – SDRAMs and Direct Rambus RAMs – on embedded systems performance has been investigated by exhaustive simulation for the first time focusing on the impact of the memory controller. Returning to the questions that have been put at the beginning of this chapter we can say:

- *RAM characteristics:* SRAMs provide constant timing and thus only require a simple memory controller. SRAMs offer superior performance but are only worthwhile for small capacities. DRAM controllers have to cope with varying access latencies depending on the inner state of the DRAM, on the order, and on the type of accesses. DRAMs however are the only choice to supply large capacities. SDRAMs can use different access granularities. SGRAMs moreover offer a bit mask that can be employed for efficient header manipulations of network packets. RDRAMs only support relatively coarse-grained data packets to access memory but they will constitute a faster solution than SDRAMs if their synchronous interface can be exploited by a suitable controller.

- *Impact of the memory controller:* The extensive exploration of operating modes of a memory controller has shown that a controller indeed has a high impact on the overall performance of a memory subsystem. Taking advantage of the pipelined interface and the row-wise organization of DRAMs has less effect on the performance of the evaluated embedded system than making use of several internal memory banks by simple address permutations. The latter operation mode alone is able to reduce the overall execution time by up to one quarter. The integration of an application-specific controller into a network processor hence is feasible and advantageous.

- *DRAM throughput vs latency properties:* The comparison of SDRAMs with RDRAMs has shown that the performance advantage of RDRAMs is rather caused by improved bus bandwidth than by any other architectural property. The use of a high throughput interface could therefore be beneficial. The access granularity of an RDRAM however is relatively coarse compared with an SDRAM. Forthcoming double data rate (DDR) SDRAMs could therefore be a more flexible solution if different or finer access resolutions were required.

Abstract RAM timing models derived from the simulation runs shown in this chapter are used in Chapter 3 for a design space exploration of our network processor.

5

Conclusion

The goal of this work is to reveal new insights into the recently emerging field of network processor design. Network processors are utilized to accelerate as well as enable tasks which are required to manage and maintain computer networks. This thesis has focused on access networks with Quality of Service (QoS) distinction among customers, applications, or even individual flows. In order to explore not only suitable algorithms for QoS preservation but also reasonable architectures for network processors, the following new methodical approaches have been applied:

- A new scheme for network services has been derived from the requirements of multi-service access networks. The scheme combines the advantages of existing services by providing quantitative guarantees for flows in terms of loss and delay, in-order delivery, graceful service degradation, and qualitative guarantees for aggregates of flows to enable resource sharing.
- A new combined evaluation of processing stages which are responsible for the QoS behavior of network access points, namely policing, queuing, and link scheduling, has been performed to study the interplay of these components that are responsible for QoS preservation.
- A design space exploration of QoS enabling algorithms and hardware resources for network processors has been carried out by co-simulation of algorithms and timing models of hardware building blocks. In this way, we have been able to appraise the usage and performance of networking resources more flexibly and exhaustively than it has been done up to now with static complexity analysis or experimental hardware platforms.
- Detailed performance models of a memory controller and synchronous DRAMs have been developed and incorporated into a mature CPU simulator. We have

been able to show how heavily the properties of a memory subsystem influence the overall computation performance of an embedded system.

Based on the above methods, the following results contribute to the field of network processor design:

- In order to implement multi-service access networks, solutions with superior timing behavior are preferred over configurations with reduced resource requirements. Setups based on Weighted Fair Queueing (WFQ) scheduling disciplines are therefore favored over solutions based on Deficit Round-Robin. This will be the right choice even if only a small number of QoS classes are supported since the delay properties can clearly be improved by using WFQ at the cost of a moderate increase in load.

- The simulation results underline that a compromise between tight delay bounds, resource requirements, and fairness properties must be found in order to implement QoS preservation policies. These three objectives cannot be optimized together. Different configurations have been suggested that optimize two of the three goals. Worst-case scenarios and inconvenient configurations for a combination of policing, queuing, and link scheduling have been determined. As a result, the policer should avoid congestion by preventively dropping packets at arrival so that the queue manager does not need to recover from congestion.

- The exploration of operating modes of a memory controller has shown that a controller indeed has a high impact on the overall performance of a memory subsystem. Taking advantage of the pipelined interface and the row-wise organization of DRAMs has less effect on the performance than making use of several internal memory banks by simple address permutations. Moreover, the performance advantage of RDRAMs compared with SDRAMs is due to the improved bus bandwidth rather than any other architectural property. It is thus feasible and pragmatic to integrate a network processor with an application-specific memory controller that supports simple operating modes and a high-bandwidth interface.

Finally, a suitable architecture for a network processor aimed at multi-service access networks has been proposed. It is characterized by a single CPU core and three distinct memory areas using different RAM types.

There are several starting-points for further research, including the following:

- It could be beneficial to feed back delay information from the architecture models to the algorithm models since an impending lack of resources should affect the behavior of a network processor. That is, besides buffer congestion, an overload of computing resources should also be considered by algorithm models.

- The QoS preservation algorithms could be adapted to support reactive flows which depend on the signalling of packet dropping and congestion events

through a feedback flow. In this case, QoS preservation requires the introduction of mechanisms for accurate reservations of the forward and feedback paths.

- In addition to the operating mode of a memory controller the careful placement of data to enhance memory performance could be taken into consideration. A suitable heuristic could be, for instance, to map parameters and variables belonging to different flows to separate internal memory banks.

A

An introduction to Petri nets

A Petri net is a formalism which can be used to model the behavior, the timing, and the structure of a system. Petri net models are used at two places in this thesis. In Chapter 4, Petri nets document the properties of a memory subsystem consisting of several components. They are used in addition for performance evaluations. In Chapter 2, a Petri net model depicts the sequence of tasks that must be performed to process a packet of information passing a network node.

A.1 Petri net basics

In the following, the basic functionality of a Petri net is defined. A Petri net is a visual formalism that not only allows to model the static properties but also the dynamic behavior of a system. More background information can be found in, for instance, [115, 128, 46].

Def. 18: (Petri net)
A Petri net *is a directed, bipartite graph* (P, T, A) *with nodes* P, T *and arcs* A*:*

P is a finite set of places,
T is a finite set of transitions,
A is a finite set of directed arcs $A \subseteq (P \times T) \cup (T \times P)$*,*
$P \cap T = \emptyset$*,*
$P \cup T \neq \emptyset$*.*

In the usual graphical representation, circles represent places, rectangles depict transitions, and directed arrows describe arcs.

Def. 19: **(Pre-set •x)** *The* pre-set $\bullet x$ *of a node* $x \in P \cup T$ *is the set of input nodes* $y \in P \cup T$ *defined by* $\bullet x = \{y | (y, x) \in A\}$.

Def. 20: **(Post-set x•)** *The* post-set $x\bullet$ *of a node* $x \in P \cup T$ *is the set of output nodes* $y \in P \cup T$ *defined by* $\bullet x = \{y | (x, y) \in A\}$.

Def. 21: **(Marking M)** *Petri net places may hold an arbitrary number of tokens. A defined distribution of tokens among the places of a Petri net is called a* marking M *that represents a system state. A marking is graphically represented by the corresponding number of dots within the places of a Petri net.*

Def. 22: **(Place-transition Petri net)**
A place-transition Petri net *is a 6-tuple* (P, T, A, C, W, M_0)*:*

(P, T, A) *is a Petri net,*
$C : P \rightarrow \mathcal{N} \cup \{\infty\}$ *is the* capacity *function that defines the maximum number of tokens hold by places,*
$W : A \rightarrow \mathcal{N}$ *defines a* weight *for each arc,*
$M_0 : P \rightarrow \mathcal{N}_0$ *is an initial marking and* $M_0(p) \leq C(p), \quad \forall p \in P$.

Weights are graphically assigned to arcs and capacities to places respectively. The default weight of an arc is one whereas the default capacity of a place is unlimited.

Ex. 12: **(Place-transition Petri net)** *Fig. 53 shows a place-transition Petri net. There are three places and three transitions. The weight of arc* (p_1, t_2) *is two and of arc* (p_2, t_3) *is three respectively. All other arcs have the default weight one. The marking is* $M(p_1) = 2$, $M(p_2) = 1$, *and* $M(p_3) = 0$. *Only the capacity of place* p_2 *is limited to 12 tokens. Some exemplary pre- and post-sets are* $\bullet t_2 = \{p_1\}$, $\bullet p_2 = \{t_1, t_2\}$, $t_1\bullet = \{p_2\}$, *and* $p_1\bullet = \{t_1, t_2\}$.

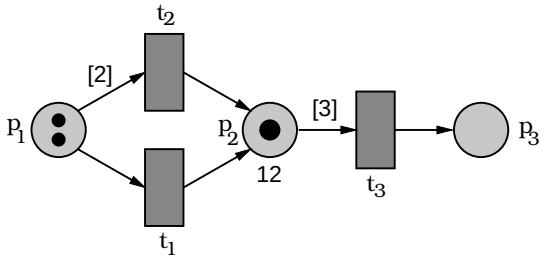

Fig. 53: An example of a place-transition Petri net.

Up to now, only the static structure of a Petri net has been described. The following definitions enable the dynamic representation of the behavior of a system. A change of the system state is modeled by a movement of tokens according to enabling and firing rules.

Def. 23: **(Enabling rule)** *A transition $t \in T$ of a place-transition Petri net is* enabled *with marking M if:*

$\forall p \in \bullet t : M(p) \geq W((p,t))$*, i.e., there are enough tokens in all input places of t, defined by arc weights,*

$\forall p \in t\bullet \setminus \bullet t : M(p) \leq C(p) - W((t,p))$*, i.e., there is enough remaining capacity for all places that are only outputs of t,*

$\forall p \in (t\bullet \cap \bullet t) : M(p) \leq C(p) - W((t,p)) + W((p,t))$*, i.e., there is enough remaining capacity for each place that is an input and an output of t.*

Def. 24: **(Firing rule)** *An enabled transition may fire. A firing of a transition t causes the following token movements in a place-transition Petri net with marking M:*

$\forall p_{in} \in \bullet t : W((p_{in},t))$ *tokens are removed from* $M(p_{in})$*,*

$\forall p_{out} \in t\bullet : W((t,p_{out}))$ *tokens are added to to* $M(p_{out})$*.*

Ex. 13: **(Token movement)** *Fig. 54 shows some possible markings derived from Ex. 12 by firing different transitions. Obviously, Ex. 12 models a conflict between transitions t_1 and t_2 since both transitions are enabled by the marking of place p_1. Which transition may fire is chosen non-deterministically. Fig. 54.a) displays a marking if transition t_2 fires whereas Fig. 54.b) represents a marking if transition t_1 fires twice. The net in Fig. 54.a) is dead since none of the transitions can be enabled by the current marking whereas in Fig. 54.b) transition t_3 could fire once.*

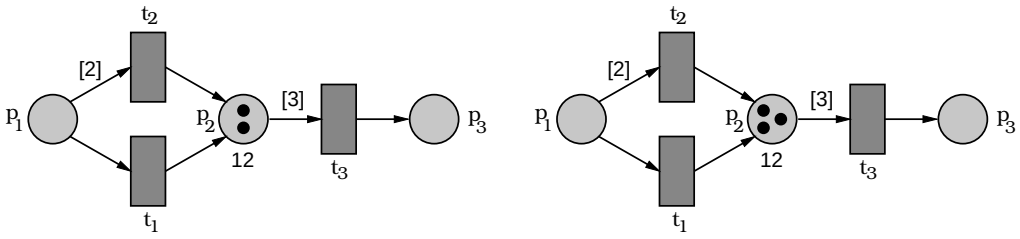

Fig. 54: Some possible markings derived from Ex. 12.

A.2 Petri net extensions

An overview of extensions to Petri nets is given. In order to understand the necessary adjustments of enabling and firing rules the reader is referred to the corresponding literature.

A so-called high level Petri net may show the following properties:

- *Colored tokens:* Tokens become distinguishable since a variable of an arbitrary data type may be assigned to a token. Enabling and firing rules may take the current values of tokens into account. An action assigned to a transition may modify the value of a token when the transition fires. An example of such a Petri net is a Coloured Petri net [91].

- *The consideration of time:* Two approaches have evolved to consider a notion of time in a Petri net model. A *timed* Petri net fires a transition as soon as it is enabled. However, a finite firing duration is assigned to each transition so that there is a delay between the removal of tokens from the pre-set and the addition of tokens to the post-set of a transition. Contrary to that, *time* Petri nets associate a finite enabling duration with each transition. Thus, a transition may fire if it is continuously enabled for at least the minimum duration and for at most the maximum duration. A time Petri net is more general than a timed Petri net, i.e., a timed Petri net can be modeled by a time Petri net. Examples of Petri net definitions that consider time can be found in [115, 91, 46].

- *Abstraction by a hierarchy of Petri nets:* Places and transitions can be refined by subnets forming a hierarchy of Petri nets. Moreover, the concept of components can be introduced to ease the reuse of Petri nets. An example of such an approach is described in [46].

The consideration of time enables the performance evaluation of a system. The use of colored tokens and hierarchical nets reduces the complexity of a model by more effectively describing the structure as well as the control and data flow of a system.

Bibliography

[1] Common Programming / Switch Interface forum (CSIX / CPIX), Network Processing Forum (NPF). *http://www.csix.org*, *http://www.cpixforum.org*, *http://www.npforum.org*.

[2] Internet Engineering Task Force (IETF). *http://www.ietf.org*.

[3] The Internet Traffic Archive. *http://www.acm.org/sigs/sigcomm/ITA/*, sponsored by the ACM Special Interest Group on Data Communication (SIGCOMM).

[4] The Moses project: Modeling, Simulation, and Evaluation of Systems, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology Zurich, Switzerland. *http://www.tik.ee.ethz.ch/∼moses*.

[5] E. Adler, J.K. DeBrosse, S.F. Geissler, S.J. Holmes, M.D. Jaffe, J.B. Johnson, C.W. Koburger, J.B. Lasky, B. Lloyd, G.L. Miles, J.S. Nakos, W.P. Noble, S.H. Voldman, M. Armacost, and R. Ferguson. The evolution of IBM CMOS DRAM technology. *IBM Journal of Research and Development*, 39(1-2):167–188, March 1995.

[6] ARM, Ltd. *ARM9E-S Technical Reference Manual, ARM DDI 0165A*, December 1999. *http://www.arm.com*.

[7] ATM Forum Technical Committee. ATM user-network interface (UNI) specification version 3.1, September 1994.

[8] Anindo Banerjea, Domenico Ferrari, Bruce A. Mah, Mark Moran, Dinesh C. Verma, and Hui Zhang. The Tenet real-time protocol suite: Design, implementation, and experiences. *IEEE/ACM Transactions on Networking*, 4(1):1–10, February 1996.

[9] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in Web client access patterns: characteristics and caching implications. *World Wide Web*, 2(1-2):15–28, 1999.

[10] Paul Barford and Mark Crovella. Generating representative Web workloads for network and server performance evaluation. *Performance Evaluation Review*, 26(1):151–160, June 1998.

[11] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *25th International Symposium on Computer Architecture*, pages 3–14, 1998.

[12] BenchWeb. Benchmark branch of the Netlib repository, University of Tennessee. *http://www.netlib.org/benchweb/*.

[13] Jon C. R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, October 1997.

[14] Jon C.R. Bennett and Hui Zhang. WF^2Q: worst-case fair weighted fair queueing. In *IEEE INFOCOM '96, The Conference on Computer Communications*, volume 1, pages 120–128, 1996.

[15] Yoram Bernet, Peter Ford, Raj Yavatkar, Fred Baker, Lixia Zhang, Michael Speer, Bob Braden, Bruce Davie, John Wroclawski, and Eyal Felstaine. A framework for integrated services operation over Diffserv networks. Request for Comments 2998, Internet Engineering Task Force (IETF), November 2000.

[16] Saleem N. Bhatti and Jon Crowcroft. QoS-sensitive flows: Issues in IP packet handling. *IEEE Internet Computing*, 4(4):48–57, July 2000.

[17] Steven Blake, David L. Black, Mark A. Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An architecture for differentiated services. Request for Comments 2475, Internet Engineering Task Force (IETF), December 1998.

[18] Keith Boland and Apostolos Dollas. Predicting and precluding problems with memory latency. *IEEE Micro*, 14(4):59–67, August 1994.

[19] Grady Booch. *Object-oriented analysis and design, with applications*. Benjamin/Cummings, 2nd edition, 1994.

[20] Bob Braden, David Clark, and Scott Shenker. Integrated Services in the Internet architecture: an overview. Request for Comments 1633, Internet Engineering Task Force (IETF), June 1994.

[21] Bob Braden, David D. Clark, Jon Crowcroft, Bruce Davie, Steve Deering, Deborah Estrin, Sally Floyd, Van Jacobson, Greg Minshall, Craig Partridge, Larry Peterson, K. K. Ramakrishnan, Scott Shenker, John Wroclawski, and Lixia Zhang. Recommendations on queue management and congestion avoidance in the Internet. Request for Comments 2309, Internet Engineering Task Force (IETF), April 1998.

[22] Bob Braden, Lixia Zhang, Steve Berson, Shai Herzog, and Sugih Jamin. Resource ReSerVation Protocol (RSVP) – version 1 functional specification. Request for Comments 2205, Internet Engineering Task Force (IETF), September 1997.

[23] Doug Burger and Todd. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin-Madison, June 1997.

[24] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *23th International Symposium on Computer Architecture*, pages 78–89, 1996.

[25] Frank P. Burns, Albert M. Koelmans, and Alex V. Yakovlev. Analysing superscalar processor architectures with coloured Petri nets. *International Journal on Software Tools for Technology Transfer, Springer-Verlag*, 2(2):182–191, 1998.

[26] Werner Bux, Wolfgang E. Denzel, Ton Engbersen, Andreas Herkersdorf, and Ronald P. Luijten. Technologies and building blocks for fast packet forwarding. *IEEE Communications Magazine*, 39(1):70–77, January 2001.

[27] Neal Cardwell, Stefan Savage, and Thomas Anderson. Modeling TCP latency. In *IEEE INFOCOM 2000*, volume 3, pages 1742–1751, 2000.

[28] John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, Lambert Schaelicke, and Terry Tateyama. Impulse: building a smarter memory controller. In *Fifth International Symposium on High-Performance Computer Architecture*, pages 70–79, 1999.

[29] Vinton G. Cerf and Robert E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, COM-22(5):637–648, May 1974.

[30] H. Jonathan Chao, Yau-Ren Jenq, Xiaolei Guo, and Cheuk H. Lam. Design of packet-fair queuing schedulers using a RAM-based searching engine. *IEEE Journal on Selected Areas in Communications*, 17(6):1105–1126, June 1999.

[31] Fabio M. Chiussi and Andrea Francini. Minimum-delay self-clocked fair queueing algorithm for packet-switched networks. In *Proceedings. IEEE INFOCOM '98, the Conference on Computer Communications*, volume 3, pages 1112–1121. IEEE, 1998.

[32] Ki-Ho Cho and Hyunsoo Yoon. Design and analysis of a fair scheduling algorithm for QoS guarantees in high-speed packet-switched networks. In *ICC '98 IEEE International Conference on Communications*, volume 3, pages 1520–1525, 1998.

[33] Yen-Ping Chu and E-Hong Hwang. A new packet scheduling algorithm: minimum starting-tag fair queueing. *IEICE Transactions on Communications*, E80-B(10):1529–1536, October 1997.

[34] Israel Cidon, Roch Guérin, and Asad Khamisy. On protective buffer policies. *IEEE/ACM Transactions on Networking*, 2(3):240–246, June 1994.

[35] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 128–137. ACM, May 1994.

[36] Jorge A. Cobb, Mohamed G. Gouda, and Amal El-Nahas. Time-shift scheduling-fair scheduling of flows in high-speed networks. *IEEE/ACM Transactions on Networking*, 6(3):274–285, June 1998.

[37] Charles D. Cranor, R. Gopalakrishnan, and Peter Z. Onufryk. Architectural considerations for CPU and network interface integration. *IEEE Micro*, 20(1):18–26, Jan.-Feb. 2000.

[38] Nicholas Cravotta. Network processors: The sky is the limit. *EDN-Magazine, US-edition*, 44(24):108–119, November 1999.

[39] Patrick Crowley, Marc E. Fluczynski, Jean-Loup Baer, and Brian N. Bershad. Characterizing processor architectures for programmable network interfaces. In *International Conference on Supercomputing (ICS) 2000*, pages 54–65. ACM, May 2000.

[40] Rene L. Cruz. A calculus for network delay. Part I. Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, January 1991.

[41] Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge. A performance comparison of contemporary DRAM architectures. In *26th International Symposium on Computer Architecture*, pages 222–233, 1999.

[42] Brian Davis, Trevor Mudge, and Bruce Jacob. DDR2 and low latency variants. In *Solving the Memory Wall Workshop at ISCA'00*, Vancouver BC, Canada, June 2000.

[43] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. *Computer Communication Review, ACM SIGCOMM*, 27(4):3–14, October 1997.

[44] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. *Internetworking: Research and Experience*, 1(1):3–26, September 1990.

[45] Robert H. Dennard. Evolution of the MOSFET dynamic RAM - a personal view. *IEEE Transactions on Electron Devices*, 31(11):1549–1555, November 1984.

[46] Robert Esser. *An Object Oriented Petri Net Approach to Embedded System Design*. PhD thesis, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Switzerland, 1996.

[47] Wenjia Fang and Larry Peterson. Inter-AS traffic patterns and their implications. In *GLOBECOM'99*, volume 3, pages 1859–1868. IEEE, 1999.

[48] Wenjia Fang, Nabil Seddigh, and Biswajit Nandy. A time sliding window three colour marker (TSWTCM). Request for Comments 2859, Internet Engineering Task Force (IETF), June 2000.

[49] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *24th International Symposium on Computer Architecture*, pages 133–143, 1997.

[50] Anja Feldmann, Anna C. Gilbert, Polly Huang, and Walter Willinger. Dynamics of IP traffic: a study of the role of variability and the impact of control. In *SIGCOMM'99, conference on Applications, technologies, architectures, and protocols for computer communication*, pages 301–313, August 1999.

[51] Anja Feldmann and S. Muthukrishnan. Tradeoffs for packet classification. In *IEEE INFOCOM 2000*, Tel-Aviv, Israel, March 2000.

[52] Paul Ferguson and Geoff Huston. *Quality of Service. Delivering QoS on the Internet and in Corporate Networks*. John Wiley & Sons, 1998.

[53] Domenico Ferrari and Dinesh C. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, 8(3):368–379, April 1990.

[54] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[55] Sally Floyd and Van Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, August 1995.

[56] Mark W. Garrett and Walter Willinger. Analysis, modeling and generation of self-similar VBR video traffic. In *SIGCOMM'94, conference on Communications architectures, protocols and applications*, pages 269–280, August 1994.

[57] Leonidas Georgiadis, Roch Guérin, and Abhay Parekh. Optimal multiplexing on a single link: delay and buffer requirements. *IEEE Transactions on Information Theory*, 43(5):1518–1535, September 1997.

[58] Leonidas Georgiadis, Roch Guérin, Vinod Peris, and R. Rajan. Efficient support of delay and rate guarantees in an internet. *Computer Communication Review*, 26(4):106–116, October 1996.

[59] Leonidas Georgiadis, Roch Guérin, Vinod Peris, and Kumar N. Sivarajan. Efficient network QoS provisioning based on per node traffic shaping. *IEEE/ACM Transactions on Networking*, 4(4):482–501, August 1996.

[60] Linda Geppert. The new chips on the block [network processors]. *IEEE Spectrum*, 38(1):66–68, January 2001.

[61] Glenn Giacalone, Tom Brightman, Andy Brown, John Brown, James Farrell, Ron Fortino, Tom Franco, Andrew Funk, Kevin Gillespie, Elliot Gould, Dave Husak, Ed McLellan, Bill Peregoy, Don Priore, Mark Sankey, Peter Stropparo, and Jeff Wise. A 200 MHz digital communications processor. In *2000 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 416–417, 2000.

[62] S. Jamaloddin Golestani. A self-clocked fair queueing scheme for broadband applications. In *INFOCOM 94*, volume 2, pages 636–646. IEEE, June 1994.

[63] Pawan Goyal and Harrick M. Vin. Generalized guaranteed rate scheduling algorithms: a framework. *IEEE/ACM Transactions on Networking*, 5(4):561–571, August 1997.

[64] Pawan Goyal, Harrick M. Vin, and Haichen Cheng. Start-time fair queuing: a scheduling algorithm for integrated services packet switching networks. *Computer Communication Review*, 26(4):157–168, October 1996.

[65] Matthias Gries. Modeling a memory subsystem with Petri Nets: a case study. In *Workshop Hardware Design and Petri Nets HWPN98 at ATPN*, pages 186–201, Lisbon, Portugal, June 1998.

[66] Matthias Gries. A survey of synchronous RAM architectures. Technical Report 71, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Switzerland, April 1999.

[67] Matthias Gries. The impact of recent DRAM architectures on embedded systems performance. In *26th Euromicro Conference on Digital System Design*, volume 1, pages 282–289, Maastricht, Netherlands, September 2000. IEEE Computer.

[68] Matthias Gries. Modeling a memory subsystem with Petri Nets: a case study. In Alex Yakovlev, Luis Gomes, and Luciano Lavagno, editors, *Hardware Design and Petri Nets*, pages 291–310. Kluwer Academic Publishers, March 2000.

[69] Matthias Gries and Jonas Greutert. Modeling a shared medium access node with QoS distinction. Technical Report 86, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Switzerland, April 2000.

[70] Matthias Gries, Jörn W. Janneck, and Martin Naedele. Reusing design experience for Petri Nets through patterns. In *High Performance Computing'99 (HPC99)*, pages 453–458, April 1999.

[71] Matthias Gries and Andreas Romer. Performance evaluation of recent DRAM architectures for embedded systems. Technical Report 82, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Switzerland, November 1999.

[72] Roch Guérin, Sanjay Kamat, Vinod Peris, and R. Rajan. Scalable QoS provision through buffer management. *Computer Communication Review*, 28(4):29–40, October 1998.

[73] Roch Guérin and Vinod Peris. Quality-of-service in packet networks: basic mechanisms and directions. *Computer Networks*, 31(3):169–189, February 1999.

[74] Pankaj Gupta, Steven Lin, and Nick McKeown. Routing lookups in hardware at memory access speeds. In *INFOCOM'98*, volume 3, pages 1240–1247. IEEE Computer and Communications Societies, 1998.

[75] Pankaj Gupta and Nick McKeown. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41, January 2000.

[76] Pankaj Gupta and Nick McKeown. Dynamic algorithms with worst-case performance for packet classification. In *IFIP Networking Conference*, Paris, France, May 2000.

[77] Linley Gwennap. Alpha 21364 to ease memory bottleneck. *Microprocessor Report*, 12(14):12–15, October 1998.

[78] Ellen L. Hahne and Robert G. Gallager. Round robin scheduling for fair flow control in data communication networks. In *IEEE International Conference on Communications'86*, volume 1, pages 103–107. IEEE, New York, NY, USA, 1986.

[79] David Harel. StateCharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[80] Juha Heinanen, Fred Baker, Walter Weiss, and John Wroclawski. Assured forwarding PHB group. Request for Comments 2597, Internet Engineering Task Force (IETF), June 1999.

[81] Juha Heinanen and Roch Guérin. A two rate three color marker. Request for Comments 2698, Internet Engineering Task Force (IETF), September 1999.

[82] John L. Hennessy and David A. Patterson. *Computer Organization & Design, the Hardware / Software Interface*. Morgan Kaufmann Publishers, 1994.

[83] Hitachi Kodaira Semiconductor Co., Ltd. *SH7750 Series Hardware Manual, ADE-602-124C*, 4 edition, March 2000.

[84] Sung I. Hong, Sally A. McKee, Maximo H. Salinas, Robert H. Klenke, James H. Aylor, and Wm. A. Wulf. Access order and effective bandwidth for streams on a Direct Rambus memory. In *Fifth International Symposium on High-Performance Computer Architecture*, pages 80–89, 1999.

[85] Nen-Fu Huang and Shi-Ming Zhao. A novel IP-routing lookup scheme and hardware architecture for multigigabit switching routers. *IEEE Journal on Selected Areas in Communications*, 17(6):1093–1104, June 1999.

[86] Geoff Huston. Next steps for the IP QoS architecture. Request for Comments 2990, Internet Engineering Task Force (IETF), November 2000.

[87] IBM Corp. Microelectronics Devision. *PowerPC 740/750 RISC Microprocessor User's Manual, GK21-0263-00*, February 1999.

[88] Intel Corp. *PC SDRAM Specification, Rev. 1.7*, November 1999.

[89] Van Jacobson, Kathleen Nichols, and Kedarnath Poduri. An expedited forwarding PHB. Request for Comments 2598, Internet Engineering Task Force (IETF), June 1999.

[90] Jörn W. Janneck. *Syntax and Semantics of Graphs: An approach to the specification of visual notations for discrete-event systems*. PhD thesis, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology Zurich, Switzerland, June 2000.

[91] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1: Basic Concepts of *EATCS Monographs in Computer Science*. Springer-Verlag, 1992.

[92] Joint Electron Device Engineering Council (JEDEC). *Standard 21C (JESD21C): Configurations for solid state memories: official and preliminary releases*.

[93] Douglas W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, April 1986.

[94] Charles R. Kalmanek, Hemant Kanakia, and Srinivason Keshav. Rate controlled servers for very high-speed networks. In *GLOBECOM '90*, volume 1, pages 12–20. IEEE, New York, NY, USA, 1990.

[95] Srinivasan Keshav. On the efficient implementation of fair queueing. *Internetworking: Research and Experience*, 2(3):157–173, September 1991.

[96] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 2 edition, 1998.

[97] Masaki Kumanoya, Toshiyuki Ogawa, Yasuhiro Konishi, Katsumi Dosaka, and Kazuhiro Shimotori. Trends in high-speed DRAM architectures. *IEICE Transactions on Electronics*, E79-C(4):472–481, April 1996.

[98] Vijay P. Kumar, T.V. Lakshman, and Dimitrios Stiliadis. Beyond best effort: router architectures for the differentiated services of tomorrow's Internet. *IEEE Communications Magazine*, 36(5):152–164, May 1998.

[99] T.V. Lakshman, Upamangu Madhow, and Bernhard Suter. Window-based error recovery and flow control with a slow acknowledgement channel: a study of TCP/IP performance. In *INFOCOM '97*, volume 3, pages 1199–1209. IEEE Comput. Soc. Press, Los Alamitos, CA, USA, 1997.

[100] T.V. Lakshman, Arnold Neidhardt, and Teunis J. Ott. The drop from front strategy in TCP and in TCP over ATM. In *INFOCOM '96*, volume 3, pages 1242–1250. IEEE Computer, March 1996.

[101] T.V. Lakshman and Dimitrios Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *Computer Communication Review*, 28(4):203–214, October 1998.

[102] Butler Lampson, Venkatachary Srinivasan, and George Varghese. IP lookups using multiway and multicolumn search. In *INFOCOM'98*, volume 3, pages 1248–1256. IEEE Computer and Communications Societies, 1998.

[103] Jean-Yves Le Boudec and Patrick Thiran. A short tutorial on network calculus. I: fundamental bounds in communication networks. In *IEEE International Symposium on Circuits and Systems (ISCAS) 2000*, volume 4, pages 93–96, 2000.

[104] Jörg Liebeherr, Dallas E. Wrege, and Domenico Ferrari. Exact admission control for networks with a bounded delay service. *IEEE/ACM Transactions on Networking*, 4(6):885–901, December 1996.

[105] Dong Lin and Robert Morris. Dynamics of random early detection. *Computer Communication Review*, 27(4):127–137, October 1997.

[106] John W. Lockwood, Naji Naufel, Jon S. Turner, and David E. Taylor. Reprogrammable network packet processing on the field programmable port extender (FPX). In *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001)*, pages 87–93, Monterey, CA, USA, February 2001.

[107] Pictro Manzoni, Paolo Cremonesi, and Giuseppe Serazzi. Workload models of VBR video traffic and their use in resource allocation policies. *IEEE/ACM Transactions on Networking*, 7(3):387–397, June 1999.

[108] Anthony J. McAuley and Paul Francis. Fast routing lookup using CAMs. In *IEEE INFOCOM'93*, volume 3, pages 1382–1391, 1993.

[109] Tony McGregor, Hans-Werner Braun, and Jeff Brown. The NLANR network analysis infrastructure. *IEEE Communications Magazine*, 38(5):122–128, May 2000.

[110] Sally A. McKee, Robert H. Klenke, Kenneth L. Wright, William A. Wulf, Maximo H. Salinas, James H. Aylor, and Alan P. Batson. Smarter memory: improving bandwidth for streamed references. *IEEE Computer*, 31(7):54–63, July 1998.

[111] Merit Networks Inc. and the University of Michigan. Internet performance management and analysis project. *http://www.merit.edu/ipma/*.

[112] Sung-Whan Moon, Kang G. Shin, and Jennifer Rexford. Scalable hardware priority queue architectures for high-speed packet switches. In *Third IEEE Real-Time Technology and Applications Symposium*, pages 203–212, 1997.

[113] Akio Moridera, Kazuo Murano, and Yukou Mochida. The network paradigm of the 21st century and its key technologies. *IEEE Communications Magazine*, 38(11):94–98, November 2000.

[114] Donald R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the Association for Computing Machinery*, 15(4):514–534, October 1968.

[115] Tadao Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[116] John Nagle. On packet switches with infinite storage. Request for Comments 970, Internet Engineering Task Force (IETF), December 1985.

[117] John B. Nagle. On packet switches with infinite storage. *IEEE Transactions on Communications*, 35(4):435–438, April 1987.

[118] Xiaoning Nie, Lajos Gazsi, Frank Engel, and Gerhard Fettweis. A new network processor architecture for high-speed communications. In *IEEE Workshop on Signal Processing Systems (SiPS) 1999*, pages 548–557, 1999.

[119] Stefan Nilsson and Gunnar Karlsson. IP-address lookup using LC-tries. *IEEE Journal on Selected Areas in Communications*, 17(6):1083–1092, June 1999.

[120] Yoichi Oshima, Bing J. Sheu, and Steve H. Jen. High-speed memory architectures for multimedia applications. *IEEE Circuits and Devices Magazine*, 13(1):8–13, January 1997.

[121] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP throughput: a simple model and its empirical validation. *Computer Communication Review*, 28(4):303–314, October 1998.

[122] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

[123] Vern Paxson. End-to-end Internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, June 1999.

[124] Mario Porrmann, Jörg Landmann, Karl M. Marks, and Ulrich Rückert. HiBRIC-MEM, a memory controller for PowerPC based systems. In *23rd Euromicro Conference: New Frontiers of Information Technology*, pages 653–657, September 1997.

[125] Charles Price. *MIPS IV Instruction Set, revision 3.1*. Mips Technologies, Inc., Mountain View, CA, USA, January 1995.

[126] Betty Prince. *High Performance Memories: New Architecture DRAMs and SRAMs - Evolution and Function, revised ed.* John Wiley & Sons Ltd., 1999.

[127] Rambus Inc. *Direct RDRAM 256/288-MBit (512K x16/18 x 32s), Preliminary information*, April 2000.

[128] Wolfgang Reisig. *Petri Nets, An Introduction*. Springer-Verlag, 1985.

[129] Yakov Rekhter and Tony Li. An architecture for IP address allocation with CIDR. Request for Comments 1518, Internet Engineering Task Force (IETF), September 1993.

[130] Jennifer L. Rexford, Albert G. Greenberg, and Flavio G. Bonomi. Hardware-efficient fair queueing architectures for high-speed networks. In *IEEE INFOCOM '96, Fifteenth Annual Joint Conference of the IEEE*

Computer Societies, volume 2, pages 638–646. IEEE Comput. Soc. Press, 1996.

[131] Lawrence G. Roberts. Beyond Moore's law: Internet growth trends. *IEEE Computer*, 33(1):117–119, January 2000.

[132] Allyn Romanow and Sally Floyd. Dynamics of TCP traffic over ATM networks. *IEEE Journal on Selected Areas in Communications*, 13(4):633–641, May 1995.

[133] Robert Rönngren and Rassul Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation*, 7(2):157–209, April 1997.

[134] Oliver Rose. Simple and efficient models for variable bit rate MPEG video traffic. *Performance Evaluation*, 30(1-2):69–85, July 1997.

[135] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta. The impact of architectural trends on operating system performance. In *15th ACM Symposium on Operating Systems Principles*, pages 285–298, 1995.

[136] James Rumbaugh, Michael Blaha, William Premerlani, and Frederick Eddy. *Object-oriented modeling and design*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1991.

[137] Hanrijanto Sariowan, Rene L. Cruz, and George C. Polyzos. SCED: A generalized scheduling policy for guaranteeing Quality-of-Service. *IEEE/ACM Transactions on Networking*, 7(5):669–684, October 1999.

[138] Robert R. Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, June 1997.

[139] Bran Selic, Garth Gullekson, and Paul Ward. *Real-time object-oriented modeling*. John Wiley & Sons Ltd., 1994.

[140] Scott Shenker, Craig Partridge, and Roch Guérin. Specification of guaranteed quality of service. Request for Comments 2212, Internet Engineering Task Force (IETF), September 1997.

[141] Scott Shenker and John Wroclawski. General characterization parameters for integrated service network elements. Request for Comments 2215, Internet Engineering Task Force (IETF), September 1997.

[142] M. Shreedhar and George Varghese. Efficient fair queuing using Deficit Round-Robin. *IEEE/ACM Transactions on Networking*, 4(3):375–385, June 1996.

[143] Keith Sklower. A tree-based packet routing table for Berkeley UNIX. In *USENIX Winter Conference*, pages 93–103, January 1991.

[144] Tammo Spalink, Scott Karlin, and Larry Peterson. Evaluating network processors in IP forwarding. Technical Report TR–626–00, Department of Computer Science, Princeton University, November 2000.

[145] Venkatachary Srinivasan, Subhash Suri, and George Varghese. Packet classification using tuple space search. *Computer Communication Review*, 29(4):135–146, October 1999.

[146] Standard Performance Evaluation Corporation. Open Systems Group, CPU benchmark suite. *http://www.spec.org*.

[147] John A. Stankovic, Marco Spuri, Krithi Ramamritham, and Giorgio C. Buttazzo. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*, volume 460 of *Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, 1998.

[148] Donpaul C. Stephens, Jon C.R. Bennett, and Hui Zhang. Implementing scheduling algorithms in high-speed networks. *IEEE Journal on Selected Areas in Communications*, 17(6):1145–1158, June 1999.

[149] Dimitrios Stiliadis. *Traffic Scheduling in Packet-Switched Networks: Analysis, Design, and Implementation*. PhD thesis, Dept. of Computer Engineering, University of California, Santa Cruz, June 1996.

[150] Dimitrios Stiliadis and Anujan Varma. Efficient fair queueing algorithms for packet-switched networks. *IEEE/ACM Transactions on Networking*, 6(2):175–185, April 1998.

[151] Dimitrios Stiliadis and Anujan Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on Networking*, 6(5):611–624, October 1998.

[152] Dimitrios Stiliadis and Anujan Varma. Rate-proportional servers: a design methodology for fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 6(2):164–174, April 1998.

[153] Ion Stoica, Hui Zhang, and T. S. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. *Computer-Communication-Review*, 27(4):249–262, October 1997.

[154] Karsten Strehl, Lothar Thiele, Matthias Gries, Dirk Ziegenbein, Rolf Ernst, and Jürgen Teich. FunState - an internal design representation for codesign. accepted for publication: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*.

[155] Subhash Suri, George Varghese, and Girish Chandranmenon. Leap forward virtual clock: a new fair queuing scheme with guaranteed delays and throughput fairness. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, 1997.

[156] Bernhard Suter, T.V. Lakshman, Dimitrios Stiliadis, and Abhijit K. Choudhury. Buffer management schemes for supporting TCP in gigabit routers with per-flow queueing. *IEEE Journal on Selected Areas in Communications*, 17(6):1159–1169, June 1999.

[157] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall Int., 3 edition, 1996.

[158] Mikkel Thorup. On RAM priority queues. *SIAM Journal on Computing*, 30(1):86–109, 2000.

[159] Henry Hong-Yi Tzeng and Tony Przygienda. On fast address-lookup algorithms. *IEEE Journal on Selected Areas in Communications*, 17(6):1067–1082, June 1999.

[160] Dinesh C. Verma, Hui Zhang, and Domenico Ferrari. Delay jitter control for real-time communication in a packet switching network. In *TRICOMM '91*, pages 35–43. IEEE, New York, NY, USA, April 1991.

[161] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high speed IP routing lookups. *Computer Communication Review, ACM SIGCOMM*, 27(4):25–36, October 1997.

[162] Kenneth M. Wilson and Kunle Olukotun. Designing high bandwidth on-chip caches. In *24th International Symposium on Computer Architecture*, pages 121–132, 1997.

[163] Tilman Wolf and Mark Franklin. CommBench - a telecommunications benchmark for network processors. In *2000 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 154–162, April 2000.

[164] Eric Wolman. A fixed optimum cell-size for records of various lengths. *Journal of the ACM*, 12(1):53–70, January 1965.

[165] Dallas E. Wrege, Edward W. Knightly, Hui Zhang, and Jörg Liebeherr. Deterministic delay bounds for VBR video in packet-switching networks: fundamental limits and practical trade-offs. *IEEE/ACM Transactions on Networking*, 4(3):352–362, June 1996.

[166] John Wroclawski. Specification of the controlled-load network element service. Request for Comments 2211, Internet Engineering Task Force (IETF), September 1997.

[167] William A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *Computer-Architecture-News*, 23(1):20 – 24, March 1995.

[168] Xipeng Xiao and Lionel M. Ni. Internet QoS: a big picture. *IEEE Network*, 13(2):8–18, March-April 1999.

[169] Hui Zhang. Service disciplines for guaranteed performance service in packet-switching networks. *Proceedings of the IEEE*, 83(10):1374–1396, October 1995.

[170] Hui Zhang and Domenico Ferrari. Rate-controlled static-priority queueing. In *IEEE INFOCOM '93. The Conference on Computer Communications*, volume 1, pages 227–236. IEEE Computer Society, 1993.

[171] Hui Zhang and Domenico Ferrari. Rate-controlled service disciplines. *Journal of High Speed Networks*, 3(4):389–412, 1994.

Acronyms

AF	Assured Forwarding PHB [80]
ALU	Arithmetic Logic Unit
API	Application Programming Interface
AS	Autonomous System
ATM	Asynchronous Transfer Mode
B-WFI	Bit Worst-case Fair Index [13]
CAM	Content Addressable Memory
CAS	Column Address Strobe (DRAM parameter)
CBR	Constant Bit Rate
CDVT	Cell Delay Variation Tolerance (GCRA parameter)
CIDR	Classless Inter Domain Routing [129]
CL	CAS Latency (DRAM parameter)
c-p	Closed-page (memory controller parameter)
CPIX	Common Programming Interface forum [1]
CPU	Central Processing Unit
CSIX	Common Switch Interface forum [1]
CYQ	Central Yellow Queue (queue manager)
CYQ enh.	Enhanced Central Yellow Queue (queue manager)
CYQ-RED	Central Yellow Queue with RED (queue manager)
DDR	Double Data Rate (RAM variant)
DiffServ	Differentiated Services [17]
DIMM	Dual In-line Memory Module [92]
DPL	Data-in to Precharge Latency (DRAM parameter)
DRAM	Dynamic RAM
DRDRAM	see RDRAM
DRR	Deficit Round-Robin [142]
DSCP	DiffServ Code Point (field in IP header)
DSL	Digital Subscriber Line
EDF	Earliest Deadline First [147]
EF	Expedited Forwarding PHB [89]
EPD	Early Packet Discard [132]
ESDRAM	Enhanced SDRAM
ESPP	Extended Simulated Protective Policy [34]
ETP	Extended Threshold Policy [34]

FCFS	First Come First Served (scheduling discipline)
FFQ	Frame-based Fair Queueing [150]
FIFO	First In First Out
FIS	Fat and Inverted Segment tree [51]
FP	Floating Point
FPGA	Field-Programmable Gate Array
FQ	Fair Queueing [44]
FRED	Flow Random Early Drop [105]
FTP	File Transfer Protocol
G.723	ITU recommendation: Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s
GCRA	Generic Cell Rate Algorithm [7]
H.263	ITU recommendation: Video Coding for low bit rate communication
HRR	Hierarchical Round-Robin [94]
HTTP	HyperText Transfer Protocol
IETF	Internet Engineering Task Force [2]
int	Interleaved address map (memory controller parameter)
Int	Integer
IntServ	Integrated Services [20]
I/O	Input / Output
IP	Internet Protocol
ISA	Instruction Set Architecture
ISO	International Organization for Standardization
ISP	Internet Service Provider
ITU	International Telecommunication Union
JEDEC	Joint Electron Device Engineering Council
JPEG	Joint Photographic Experts Group
LAN	Local Area Network
LC	Level Compressed trie [119]
LFVC	Leap Forward Virtual Clock [155]
LIFO	Last In First Out
lin	Linear address map (memory controller parameter)
LQD	Longest Queue Drop [156]
MAN	Metropolitan Area Network
MD-SCFQ	Minimum-Delay Self-Clocked Fair Queueing [31]
MMU	Memory Management Unit
MOSES	MOdeling, Simulation, and Evaluation of Systems [4]
MPEG	Moving Picture Experts Group
MSFQ	Minimum Starting-tag Fair Queueing [33]
NNTP	Network News Transfer Protocol
NPF	Network Processing Forum [1]
NTSC	National Television Standards Committee

o-p	Open-page (memory controller parameter)
OSI	Open Systems Interconnection reference model of ISO
PAL	Phase Alternation Line (television display standard)
PBSRAM	Pipelined Burst SRAM
PC	Personal Computer
PCR	Peak Cell Rate (GCRA parameter)
PHB	Per Hop Behavior [17, 80, 89]
PISA	Portable ISA [23]
QM	Queue Manager
QoS	Quality of Service
RAM	Random Access Memory
RAS	Row Address Strobe (DRAM parameter)
RCD	RAS to CAS Delay (DRAM parameter)
RDRAM	(Direct) Rambus DRAM
RED	Random Early Detection [54]
RfC	Request for Comments (IETF document status)
ROOM	Real-time Object-Oriented Modeling [139]
RP	RAS Precharge time (DRAM parameter)
RPS	Rate-Proportional Server [152]
RR	Round-Robin (scheduling discipline)
RSVP	Resource ReSerVation Protocol [22]
SCFQ	Self-Clocked Fair Queueing [62]
SDRAM	Synchronous DRAM
SFQ	Start-time Fair Queueing [64]
SGRAM	Synchronous Graphic RAM
SLA	Service Level Agreement
SoC	System-on-a-Chip
SP	Static Priority (scheduling discipline)
SPEC	Standard Performance Evaluation Corp. [146]
SPFQ	Starting Potential-based Fair Queueing [150]
SRAM	Static RAM
TCP	Transmission Control Protocol
ToS	Type of Service (field in IP header)
trie	Data structure for reTRIEval
TSpec	Traffic Specification [141]
T-WFI	Time Worst-case Fair Index [14]
UDP	User Datagram Protocol
VBR	Variable Bit Rate
VC-SDRAM	Virtual-Channel SDRAM
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VPN	Virtual Private Network
VTRR	Virtual Time-based Round-Robin [32]

WAN	Wide Area Network
WAR	Write-After-Read (memory access sequence)
WFQ	Weighted Fair Queueing [122]
WF^2Q	Worst-case Fair WFQ [14]
WWW	World Wide Web
YQ-Fair	Fair Yellow Queue (queue manager)
ZBT	Zero Bus Turnaround (RAM operating mode)