Diss. ETH No. 15093

System-Level Timing Analysis and Scheduling for Embedded Packet Processors

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
Doctor of Sciences

presented by
SAMARJIT CHAKRABORTY
M.Tech. Computer Science & Engg.,
Indian Institute of Technology Kanpur, India
born December 25, 1972
citizen of India

accepted on the recommendation of
Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Rolf Ernst, co-examiner

2003

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. 54

Samarjit Chakraborty

System-Level Timing Analysis and Scheduling for Embedded Packet Processors

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A dissertation submitted to the
Swiss Federal Institute of Technology Zurich
for the degree of Doctor of Sciences

Diss. ETH No. 15093

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Rolf Ernst, co-examiner

Examination date: April 14, 2003

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. 54

Samarjit Chakraborty

System-Level Timing Analysis and Scheduling for Embedded Packet Processors

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Diss. ETH No. 15093

System-Level Timing Analysis and Scheduling for Embedded Packet Processors

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
Doctor of Sciences

presented by
SAMARJIT CHAKRABORTY

M.Tech. Computer Science & Engg.,
Indian Institute of Technology Kanpur, India
born December 25, 1972
citizen of India

accepted on the recommendation of
Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Rolf Ernst, co-examiner

2003

Examination date: April 14, 2003

Abstract

Packet processors are high-performance, programmable devices with special architectural features that are optimized for network packet processing. They are mostly embedded within network routers and switches and are designed to implement complex packet processing tasks at line speed.

In this thesis we study several issues related to the system-level timing analysis and scheduling for such embedded packet processors. Our work is motivated by the fact that designing and analysing, both, the architecture and the software for packet processors require new models and methods which do not fall within the preview of traditional embedded systems design.

Both, timing analysis and scheduling have been widely studied in the context of system-level design of embedded systems. But most of these studies have largely focussed on data-dominated applications like digital signal processing and developed techniques which do not directly hold for control-dominated applications like packet processing. We identify the main differences between these two classes of applications, and also the architectures used in these two cases, and based on these differences propose new models for tasks and architectures which are especially suited to capture the characteristics of control-dominated applications and focus on one particular class, viz. network packet processing. Based on these models, we then propose several algorithms for analysing the hardware/software-architectures of packet processors. In this context, the different problems addressed in this thesis and our main contributions can be summarized as follows.

- Packet processors usually consist of a collection of heterogeneous processing elements. We pose the problem of determining the feasibility of a mapping of the different packet processing tasks onto the different processing elements, as a schedulability analysis problem. It turns out that for the model we consider, this schedulability analysis problem is intractable (NP-hard) and therefore can not be solved within any reasonable time. To get around this, we introduce a novel concept called "approximate schedulability analysis", using which the problem can be solved in polynomial time if a small error in the decisions made by the algorithm is allowed. Using this concept, we demonstrate that in spite of the intractability result, a schedulability analysis can nevertheless be done in reasonable time for all practical purposes. We also show that this concept is not only restricted to our particular model in the context of packet processing, but is applicable to a wide variety of other real-time task models for which only

exponential or pseudo-polynomial time algorithms were known till now.

- We study an analytical framework for system-level timing analysis for packet processors, which generalizes many scheduling-theoretic results from the real-time systems area, and also matches results that could previously be obtained only using detailed cycle-accurate simulations. Based on this, we propose a new methodology for the design space exploration of packet processors to tackle the large design space involved. It is based on conducting the exploration in several stages, each at a different level of abstraction, and using a different performance evaluation scheme in each stage.

- Traffic management is one of the main functions of any packet processor, especially in the case of routers, where the goal is to meet the real-time constraints of QoS sensitive flows and at the same time provide a reasonable service to best-effort packets. In this context we propose a novel scheduler which gives theoretical guarantees on the service that can be provided to best-effort flows. The theoretical framework behind this scheduler generalizes a number of service schemes developed in the real-time systems area for integrating soft-real-time jobs into a hard-real-time environment. Further, our experimental results also suggest clear improvements over previously known schemes.

The above problems are concerned with three very general issues related to scheduling and timing analysis, which arise in many different real-time embedded system scenarios. Given a set of jobs with a set of constraints on these jobs, and a goal (such as deadlines) to be met, the first problem asks "*does there exist an execution order or schedule* for the jobs which satisfies the constraints and meet the specified goal?" The second problem is concerned with answering "*given a schedule* or an execution order for the jobs, *what timing properties do the jobs satisfy* if they are executed according to this schedule?" Finally, the third problem is concerned with "*finding a schedule* for the jobs which satisfies the constraints and meets the goal". The results corresponding to these problems that we derive in this thesis either extend or generalize previously known results from the real-time systems area, and also integrate concepts from scheduling theory, system-level design, and computer networking.

Kurzfassung

Paketprozessoren sind programmierbare Hochleistungsbausteine mit speziellen Architekturmerkmalen; sie sind für Paketverarbeitung in Datennetzen optimiert und meistens in Netzrouter und -switches eingebettet. Paketprozessoren werden für die verzögerungsfreie Implementation komplexer Paketverarbeitungsaufgaben entworfen.

In dieser Dissertation werden verschiedene Aspekte der Timing-Analyse und des Scheduling für solche eingebetteten Paketprozessoren auf Systemebene untersucht. Die Arbeit ist motiviert durch die Tatsache, dass der Entwurf und die Analyse sowohl der Architektur als auch der Software für Paketprozessoren neue Modelle und Methoden erfordern, welche nicht Teil des traditionellen Entwurfs eingebetteter Systeme sind.

Sowohl Timing-Analyse als auch Scheduling sind im Kontext des Entwurfs von eingebetteten Systemen auf Systemebene umfassend untersucht worden. Die meisten dieser Untersuchungen haben sich jedoch auf datendominierte Anwendungen wie digitale Signalverarbeitung konzentriert, und Techniken entwickelt welche nicht direkt in kontrolldominierten Anwendungen wie der Paketverarbeitung anwendbar sind. Wir identifizieren die Hauptunterschiede zwischen diesen beiden Anwendungskategorien und der entsprechenden Architekturen; auf diese Unterschiede gestützt, schlagen wir neue Modelle für Tasks und Architekturen vor, welche besonders geeignet sind, die Merkmale kontrolldominierter Anwendungen zu erfassen. Dabei konzentrieren wir uns auf eine spezielle Anwendungskategorie, nämlich Paketverarbeitung in Datennetzen. Auf der Grundlage dieser Modelle schlagen wir dann mehrere Algorithmen zur Analyse der Hardware-/Software-Architekturen von Paketprozessoren vor. In diesem Zusammenhang können die verschiedenen in dieser Dissertation behandelten Fragestellungen und unsere Hauptbeiträge wie folgt zusammengefasst werden:

- Paketprozessoren bestehen typischerweise aus einer Ansammlung von heterogenen Verarbeitungselementen. Wir behandeln dabei das Problem, die Machbarkeit einer Zuordnung der verschiedenen Paketverarbeitungsaufgaben zu den verschiedenen Verarbeitungselementen zu ermitteln, als Zeitplanbarkeitsanalyseproblem. Es stellt sich heraus, dass dieses Problem für das von uns betrachtete Modell unlösbar (NP-hart) ist und daher nicht in vernünftiger Zeit gelöst werden kann. Um dieses Problem zu umschiffen, führen wir ein neues Konzept namens “approximative Zeitplanbarkeitsanalyse” ein, mit welchem das Problem in polynomieller Zeit gelöst werden kann, sofern ein kleiner Fehler in den vom Al-

gorithmus getroffenen Entscheidungen erlaubt wird. Unter Zuhilfenahme dieses Konzepts zeigen wir, dass trotz des Unlösbarkeitsresultats eine Zeitplanbarkeitsanalyse dennoch für alle praktischen Zwecke in vernünftiger Zeit gemacht werden kann. Wir zeigen auch, dass dieses Konzept nicht auf unser besonderes Modell im Kontext der Paketverarbeitung beschränkt ist, sondern auf eine Vielzahl anderer Echtzeit-Taskmodelle angewendet werden kann, für die bislang nur Algorithmen mit exponentieller oder pseudo-polynomieller Laufzeit bekannt waren.

- Wir untersuchen ein analytisches Rahmenwerk für die Timing-Analyse von Paketprozessoren auf Systemebene, welches viele Resultate aus der Zeitplanungstheorie im Bereich der Echtzeitsysteme verallgemeinert; weiterhin stimmt es mit Ergebnissen überein, die zuvor nur mit detaillierten zyklengenauen Simulationen erzielt werden konnten. Basierend hierauf schlagen wir eine neue Methodologie für die Exploration des Entwurfsraums von Paketprozessoren vor, mit der der grosse Entwurfsraum in Angriff genommen werden kann. Sie gründet auf der stufenweisen Durchführung der Exploration, wobei auf jeder Stufe eine verschiedener Abstraktionsgrad und ein anderes Leistungsevaluationsschema zur Anwendung kommt.

- Die Netzverkehrsabwicklung ist eine der Hauptfunktionen jedes Paketprozessors, besonders im Fall von Routern, bei denen das Ziel in der Erfüllung der Echtzeitvorgaben QoS-empfindlicher Datenflüsse und der gleichzeitigen, vernünftigen Abarbeitung von Best-Effort-Paketen besteht. In diesem Zusammenhang schlagen wir einen neuen Scheduler vor, welcher theoretische Garantien auf die für Best-Effort-Datenflüsse zur Verfügung stehende Leistung gibt. Das theoretische Rahmenwerk hinter diesem Scheduler verallgemeinert eine Reihe von Service-Schemata, die im Bereich der Echtzeitsysteme für die Integration von Soft-Real-Time-Jobs in eine Hard-Real-Time-Umgebung entwickelt wurden. Weiterhin legen unsere experimentellen Ergebnisse deutliche Verbesserungen gegenüber zuvor bekannten Schemata nahe.

Die obigen Fragestellungen behandeln drei sehr allgemeine Themen im Zusammenhang mit Scheduling und Timing-Analyse, welche in vielen verschiedenen Echtzeitszenarien mit eingebetteten Systemen auftauchen. Gegeben eine Menge von Jobs mit einer Menge von Einschränkungen sowie ein zu erreichendes Ziel (etwa Deadlines), lautet die erste Frage “*Gibt es eine Ausführungsreihenfolge* für die Jobs, welche die Einschränkungen erfüllt und das gesetzte Ziel erreicht?” Die zweite Frage lautet “*Gegeben eine Ausführungsreihenfolge* für die Jobs, *welche Timing-Eigenschaften erfüllen die Jobs bei Ausführung gemäss der vorgegebenen Reihenfolge?*” Zuletzt beschäftigt sich die dritte Fragestellung mit dem “*Finden einer Ausführungsreihenfolge* für die Jobs, welche die Einschränkungen erfüllt und das gesetzte Ziel erreicht”. Die Antworten auf diese Fragestellungen, die wir in dieser Dissertation herleiten, erweitern oder verallgemeinern zuvor gefundene

Ergebnisse aus dem Bereich der Echtzeitsysteme, und integrieren Konzepte der Zeitplanungstheorie, des Entwurfs auf Systemebene, und der Datennetze.

Contents

1

Introduction

The primary components of any communication network are *hosts* and *routers*. Applications running on different hosts communicate with each other by sending *packets*. Any large network, including what is referred to as the *Internet* is organized in a hierarchical manner. At the lowest level, a number of hosts are connected together to form a *network*. Routers are hosts within such a network, which have interfaces to more than one network. A packet originating from an application running on a host and destined to a host in a different network, arrives at a router and is forwarded by it to the appropriate network on the basis of the destination address in the packet's header. A number of networks connected together by routers in such a manner form a larger network and the next level of hierarchy.

Routers, even with this basic "store-and-forward" functionality can be considered as "packet processors", and this is still their default behaviour in IP (or the Internet Protocol based) networks. However, with networks extensively growing in size, and the Internet slowly shifting from a research network into one being used for commerce, banking, communication, entertainment and information dissemination, routers became more and more complex and incorporated new packet processing functionality. Functions implemented within a router now include firewalls, network address translaters, means for implementing quality-of-service (QoS) guarantees to different packet flows, and also pricing mechanisms. Until recently, such a router used to be implemented entirely in software, running on a general purpose processor within a host computer. However, such implementations are increasingly becoming infeasible because of two reasons. Firstly, during the last couple of years the available network bandwidth has been on the rise, and with the advent of optical fibers being deployed for networking, network bandwidth has increased exponentially. This has lead to very stringent performance requirements from routers since they have to process packets at line speed. Secondly, in spite of the increased usage of the Internet, its structure is still relatively simple, with the underlying network providing basic communication between end-systems. All the complex services and features are therefore either implemented on the end-systems (such as those pertaining to guaranteed communication), or on the routers (such as features which need network support and can not be implemented solely by the end-systems). Additionally, any new service to be supported by a network is implemented by extending or modifying the routers.

The real-time packet processing constraints imposed on routers to support high line speeds indicate hardware-based solutions, where the router functionality is implemented on application-specific integrated circuits (ASICs). The requirements for flexibility and the complex nature of many of the processing functions, on the other hand, favour software based implementations on general purpose processors. To address these two conflicting issues, recently a new class of devices called *network processors* have emerged. These are high-performance, programmable devices with special architectural features that are optimized for packet processing.

Such specialized packet processors for implementing router functionality are commercially being developed only very recently. There is still a lot of effort to be spent in quantitatively understanding many system issues related to the architecture of these devices and also those related to designing the software running on them. This thesis analyses a number of such issues pertaining to timing analysis and scheduling for such packet processors, and towards this proposes appropriate models and algorithms. It was motivated by the fact that designing and analysing, both, the architecture for such processors and the associated software requires new models and methods which do not fall under the domain of traditional embedded systems design. Throughout this thesis, the term “packet processor” is used in the most generic sense and is supposed to encompass all kinds of network processors, task-specific packet processors, and routers made up of both network processors and general purpose processors.

1.1 Packet processing functions

There are several possibilities of classifying network packet processing functions, each providing a different insight into the requirements from the architecture on which such functions may be implemented. In this section we briefly describe such functions that may be implemented within an IP router, and three approaches towards classifying them. The first is based on the different packet processing tasks constituting a packet processor and the sequence in which such tasks process a packet, the second approach is based on the functionality of the different protocols and packet processing functions, and the third is on the basis of different services/protocols that routers might be required to support.

1.1.1 Processing task chains

In any IP based network, every network interface has a 32-bit identifier called the *IP address*. Host computers with more than one network interface have more than one IP address. To send data from a source computer to a destination host accross the Internet, the source splits the data into blocks and each such block forms the data portion of a packet. To each such block is attached an *IP header* which contains all the routing information required to send the block from the source to the destination in a hop by hop manner, by traversing several hosts in the network. The data block together with the header is called an *IP packet*. The packet is encapsulated into a link-layer packet (for example, as an Ethernet frame) such that it is appropriate for transmission over the network.

For guaranteed delivery of the packet, on each host, that the packet traverses on its way from the source to the destination, different packet processing algorithms and protocols are implemented. These algorithms implement packet processing starting at the network layer of the Transmission Control/Internet Protocol (TCP/IP) stack. This is the lowest layer from which end-to-end packet transmission between two hosts is distinguishable.

With the packet arriving at a router, a decision is to be made concerning the network interface to which the packet will be forwarded. This decision is based on the information stored in the IP packet header and also several state informations in the router. Additionally, all packets undergo some amount of transformation when they pass through the router. The minimum amount of processing that a packet might undergo is decrementing the time-to-live (or TTL) field in the packet header and recomputing the header checksum. But many packets would undergo more complex processing than this. Lastly, there might be packets which are not to be forwarded and are meant for the router itself (for example, a routing protocol packet). What is referred to as "packet processing" includes all the tasks involved in the above mentioned processing.

Since all packets passing through a router do not require the same processing, it is meaningful to divide the entire packet processing functionality of a router into a number of separate tasks. Depending on the performance and the QoS requirements of the packets to be processed, the different tasks may be distributed over multiple heterogeneous processors. Apart from making software development easy and facilitating reuse, this view of splitting packet processing into a number of tasks also opens up different implementation possibilities which might have significant impact on performance. The sequence in which different tasks process an incoming packet results in a packet processing task chain. If such chains for several packet *flows* are combined, the resulting task graph might have branches when packets from different flows require to be processed by different tasks (as an example, see Figure 1).

Basic packet processing tasks at a router include header parsing, packet classification to assign the packet a QoS class, determination of the outgoing network interface (i.e. forwarding), checking *Service Level Agreements* (i.e. policing), queuing, and link scheduling. Which among these tasks process any given

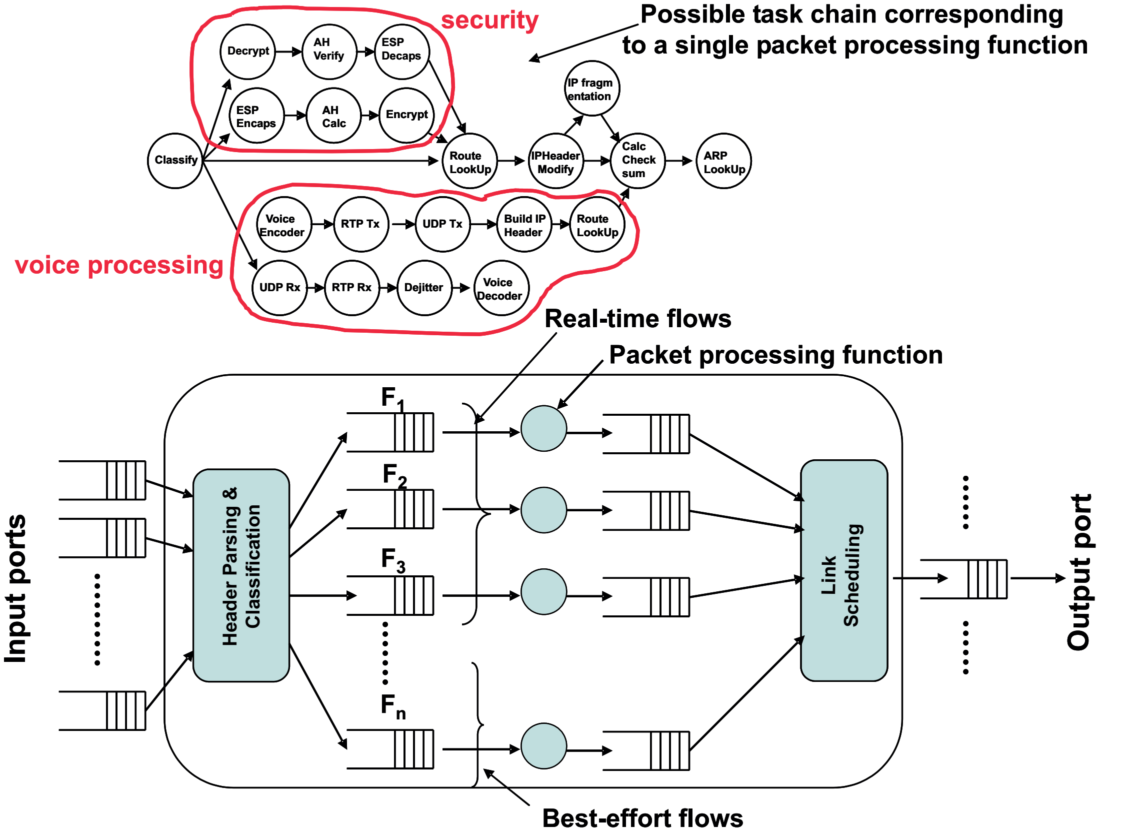

Fig. 1: Tasks in a packet processor. After the basic classification, a flow might still be an aggregate of several flows which are classified in later stages. Each node labeled as “packet processing function” can be made up of several tasks, resulting in a conditional task graph. Such a conditional task graph corresponding to one node is shown in the top of the figure.

packet, and how complex each task is, depends on the services that the router implements. Below, we describe each of these basic tasks and briefly discuss their possible manifestations and the range of their complexity.

- *Header parsing*: The header of an incoming packet is parsed to extract information based on which further processing of the packet is carried out. The information extracted might include source and destination addresses, checksums, packet length, protocol specifiers and the type of service the packet is destined to receive. It may be noted that the header parsing need not be limited to the network layer header and might include headers of higher layers of the protocol stack. As an example, IP routers often use the source and destination port numbers of the transport layer to classify packets. Based on the outcome of the header parsing, a packet may be immediately dropped, or passed on to further processing stages. In the later case, a classification task uses the extracted header information to assign the packet its context information, such as the corresponding QoS class.

- *Classification and routing*: Based on the destination address and other information extracted by the header parsing task, the packet is either forwarded to an appropriate outgoing link, or is passed to other processing tasks. A packet needs to be processed by other processing tasks either if the destination of the packet is reached, or if some higher protocol layers need to be processed. This additional processing might be implemented by several sequential tasks, and packets from different flows might pass through different task chains. Functions implemented here might include access control (by blocking certain packets from entering the network), network address translation, QoS differentiation (where real-time traffic might be separated from best-effort traffic and treated separately), policy-based routing, etc.

  Since it is possible to distinguish between different packet flows only after the header parsing and packet classification stages, both these tasks should be implemented such that they are able to process packets at line speed. This is to ensure that packets from high-priority flows receive a preferential treatment over lower priority flows during later processing/link scheduling stages. Depending on which implementation options are available, full packet clasification need not be done at this stage itself. What is implemented as “classification” at this stage, might be limited to the processing that can be completed within a fixed number of processor cycles, which is determined by the line rate. To implement application-level classification and processing, which might take an arbitrary length of time, an incoming packet might only be partially classified at the first stage. After the preliminary QoS distinction is done, packets from lower QoS classes might even be dropped if there are not sufficient processing resources available.

- *Policing*: Packets to be forwarded to the outgoing link are processed by a task which implements policing. Based on the flow information assigned by the classifier, the policer checks whether a packet conforms to the specified *traffic profile* for that flow. Traffic profiles are determined by Service Level Agreements (SLAs) between customers and the service provider for a flow, and specifies properties like the maximum allowable burstiness and the rate of incoming traffic. If a packet conforms to the specified profile, then it is guaranteed a certain minimum amount of service and is processed further without any restriction. However, if a packet does not conform to the specified profile, then the amount of service it gets might depend on the resources (i.e. it is given a “best-effort service”), or it might even be dropped. In many cases, a packet might be delayed before the policing stage, to shape the flow to which the packet belongs to the predefined profile. Details about specifying traffic profiles and shaping are described in detail in the next chapter.

- *Queuing and link scheduling*: Before a packet is finally put out into the outgoing link, it is queued until the link scheduler chooses it for transmission or is dropped in case the link is congested. Since the space for storing such packets is usually limited, it is the job of the *queue manager* to manage the packet storage space, and isolate packets from different flows which have different QoS requirements (as shown in Figure 1).

Finally, the link scheduler decides the ordering among the packets stored in the queue(s) for transmission into the outgoing link. This scheduling may be based on factors such as real-time constraints of the different flows, the traffic profiles associated with the flows, fairness requirements, efficiency requirements of the scheduler, etc.

Clearly, the exact nature and the complexity of each of the above tasks vary from case to case, and to a large extent is also constrained by the implementation choices available. As already mentioned, each of these tasks also have different performance requirements. The hardware and the software implementation of a packet processor therefore involves several tradeoffs between cost, efficiency and flexibility.

1.1.2 Control- versus data-plane

From a functionality perspective, packet processing can be divided into two classes—functions constituting the *control-plane* of a router and those constituting the *data-plane*. Each of these two classes has different characteristics and performance requirements. Typically, control-plane functions and protocols are complex and have long code paths, but do not have very high performance requirements or real-time constraints (i.e. they need not be executed at the full line speed). Examples of these include the resource reservation protocol (RSVP) which is used to allocate resources in routers for IP flows and the open shortest path first (OSPF) protocol which is used to establish and update routing tables. Generally, control-plane functions are concerned with the overall system management and are best implemented on general purpose processors because of their inherent complexity.

Data-plane functions and protocols are mostly responsible for packet forwarding. These are performance critical since they must be performed at high speed to avoid packet dropping, and to meet QoS requirements. Further, they typically require the same functionality to be applied to a large number of packets, with each packet getting a small amount of processing time. Therefore, such functions are suited for parallel execution and may be implemented on specialized processors since they are generally simple and have short code paths. Many currently available network processors provide special hardware support for common data-plane functionality. A common solution is to have a data-plane processor with a number of multi-threaded packet processing engines, which offer the required parallelism for implementing data-plane functions. For example, Intel's IXP family of network processors consist of eight to sixteen packet processing engines (commonly referred to as "microengines"), each of which contains eight hardware threads, special queues for receiving and transmitting packets, hardware state machines that specifically read from and write to the transmit and receive queues, and other hardware elements.

The distinction between data- and control-plane protocols becomes fuzzy as we move up the TCP/IP protocol stack. As an example, the transmission control protocol (TCP), which is a *Transport layer* protocol (responsible for end-to-end transmission of aggregated packets), exibits both control- and data-plane functionality and is relatively complex. As a result, designing packet processors for implementing functions and protocols belonging to higher levels of the protocol stack becomes relatively complex.

1.1.3 Services and protocols

Lastly, we list below a number of services and protocols, which require network packet processing at the routers for their implementation (i.e. they can not be implemented solely by applications running on end-systems). Here we give a high-level view of these services. Typically, they would be implemented by a sequence of tasks (as described in Section 1.1.1), some of which would be a part of the control-plane of a router and others need to be implemented in the data-plane.

- *Firewalls*: These are components used to block the flow of packets between two networks and in the process implement security features. The filtering rules used to block packets (where certain classes of packets might be selectively blocked), either from or into the network, are defined by network administrator and might be numerous and very complex. They might also involve examining the packet body (or payload) and therefore be computationally intensive. Since, in many cases, this processing has to be done in line speed, implementing firewalls might be complex.

- *Network address translators (NATs)*: Due to the limitation in the number of IP addresses available (an IP address being only 32-bits wide), NATs allow multiple hosts in an internal network to appear as a single host to the outside world by using a single IP address. Packets passing between such an internal network and the Internet are modified by the NAT. A router supporting NAT modify the source address of outbound packets and the destination address of inbound packets.

- *Transcoding media gateways*: With the increase in the number of Internet-connected devices such as pagers, cellular phones, and personal digital assistants (PDAs)—many of them of them supporting streaming audio and video—there is a need for efficiently delivering web-based content to such devices. Since most of these devices have constraints on network bandwidth and also processing capability, they are unable to receive and process data that a normal computer would be able to do. Transcoding media gateways are specialized routers that convert such high-bandwidth and high-resolution data into lower bandwidth (which might be possible to transfer over wireless links, for example) or lower resolution data (which might be suitable for, say, handheld displays).

It is possible to enlist several such services that are supported by networks today. The routers in question therefore have to perform complex packet pro-

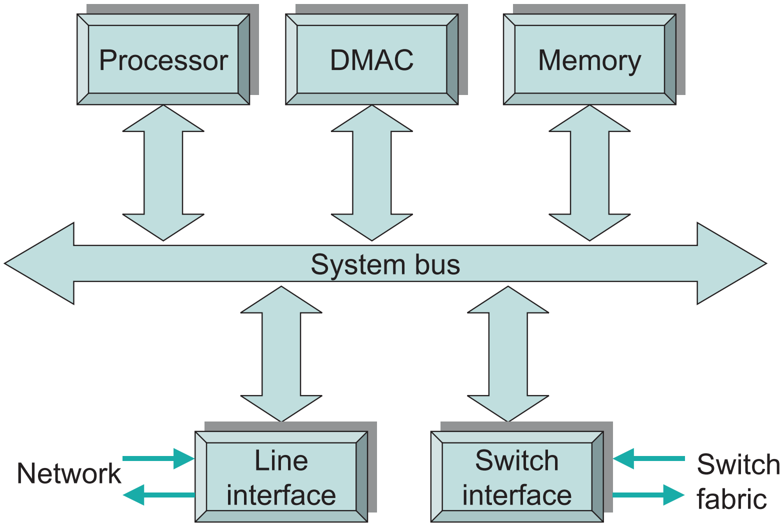

Fig. 2: Software based packet processor, built around a single general purpose processor.

cessing tasks. For hardware based solutions, as new services get introduced, the hardware needs to be updated, or the new services need to be implemented in software. Purely software based solutions on the other hand do not meet up to the stringent performance requirements enforced by the high line speeds. The present trend, as mentioned in begining of this chapter, is to have specialized devices to perform packet processing—which are programmable and at the same time some of the tasks are implemented in hardware. In the next section we briefly review the development of this class of devices.

1.2 Packet processing hardware

As mentioned in the begining of this chapter, early routers were built around a general purpose processor inside a host computer. Even today, such purely software based routers are commonly used and provide an economical and flexible way of adding router functionality to a server. From a hardware perspective, in these systems there is no differentiation between control- and data-plane functions (or what is also referred to as *slow path* and *fast path* functions)—both being handled by a single processor. The architecture of such a system is shown in Figure 2. There are interfaces to the network and the switch fabric. The general purpose processor is supported by a direct memory access controller (DMAC) and simple input/output devices. Packets are transferred between the memory and the switch interface or the line interface. The processor accesses the packets from the memory, processes them and programs the peripheral devices to transfer them to the outgoing link.

With the increase in network bandwidth and the complexity of the packet processing functions, such architectures reached their performance and scalability limits. To cope with this problem, as a first solution, some of the router

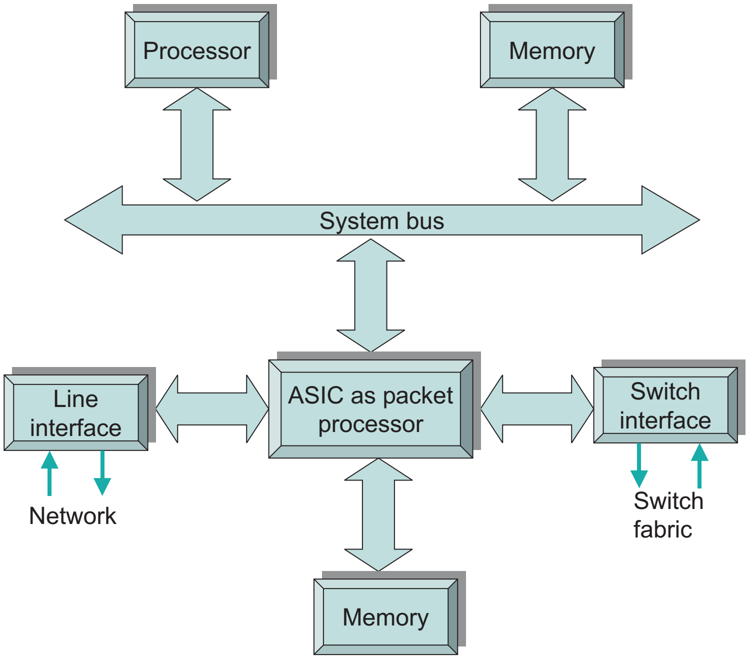

Fig. 3: A packet processor with the fast-path packet processing implemented in ASICs.

functionality was distributed to the line interface cards. The host processor handled functions related to system control and the network management interface, which are relatively time-insensitive. Processors on the line cards performed the time-critical functions. The exact distribution of functions were largely dependent on vendor specific implementations, but distributing functions among multiple processors became the common way to acclerate packet processing. The second solution was to use Ethernet switches. These are high-performance, multiport versions of Ethernet bridges, which forward frames based on layer-2 MAC address information. Using Ethernet switches at the edges of a LAN, and routers at the network core, results in traffic going to a router only if the destination address is unknown or outside the LAN. By eliminating all the complex processing required to route traffic, Ethernet switches could achieve high throughput by implementing packet processing in application-specific integrated circuits (ASICs), thereby avoiding software execution in the fast path.

It was natural to also combine the two above solutions, resulting in layer-2 switches that can perform routing, or routers with switching line cards. The architecture of such a system is shown in Figure 3, where most of the packets can be transferred without passing over the system bus and through the general purpose processor. In such systems, ASICs handled all the layer-2 processing in the fast path. The general purpose processor, on the other hand, handled all the packet processing at layer-3 and higher layers.

With the advent of layer-3 switching, the distinction between switches and routers became even more fuzzy. Because of the standardization of the of IP as the layer-3 protocol in local- and wide-area networks, layer-3 switching made it possible to do classification and forwarding of packets based on their layer-

3 address, without resorting to complex routing algorithms. Since IP packets make up the major portion of traffic in any switch or router, many system vendors offered solutions where IP classification and forwarding were hardwired into ASICs. These represented a simpler subset of more complex routing code that were executed in general purpose processors.

This line of development, however, did not stop at layer 3 in terms of the depth of packet classification. With the growth of the World Wide Web and the standardization of the associated protocols, it became possible to do even deeper packet classification and processing in the fast path, based on information available at layer 4 (such as the TCP port number) and layer 5 (such as the URL address). As a result, such processing also became candidates for ASIC-based implementation.

1.2.1 Programmable packet processors

With more and more packet processing functions moving up the protocol stack, a greater degree of variability was encountered relative to the layer-2 and layer-3 specifications. As a result, ASIC-based implementations faced limitations when it came to adding new protocols and applications, or modifying old ones. Web server switches and IP service platforms are two good examples which had serious problems with ASIC-based implementations.

The flexibility limitations of ASIC-based implementations and the performance bottlenecks of purely software-based implementations led to the development of “programmable packet processors”, which are commonly referred to as “network processors”. The architecture of network processor based line card is primarily characterized by replacing the fixed-function “ASIC as packet processor” in Figure 3, with one or more programmable processors along with specialized coprocessors for certain tasks.

1.2.1.1 Architectural characteristics

The main architectural characteristics of network processors over general purpose processors or ASICs are the following.

Firstly, the programmable processor cores in most network processors are based on modified versions of standard RISC instruction sets. The degree of modification, however, varies from product to product. Usually there are additional instructions which are tailored to speed up operations which might appear in portions of the application code having real-time constraints. Examples of these are bit manipulation instructions and instructions for searching and addressing specialized data structures.

Secondly, the programmable processor cores are supported by additional hardwired function blocks, so that the performance of functions which are common across many applications are acclerated. Examples of this are dedicated processors for encryption and decryption. Thirdly, to exploit the parallelism in terms of independently processing packets belonging to different flows, most network processor architectures have appropriate features. Parallel and pipelined processors allow packets from different flows to be processed independently, thereby allowing more processing time per packet. Lastly, packet processing typically involves a number of table lookups to match fields in the packet against values stored in lookup tables, and also buffer management as packets usually have to be stored before and after processing. To effectively support these, packet processors usually have distributed and shared memory architectures.

To give an example of the above characteristics—the Intel IXP line of processors have a general purpose 32-bit RISC processor for slow-path processing and control plane operations. Additionally, there are a number of RISC processing engines called *microengines* for fast-path processing (the IXP2400, for example, has eight microengines), where each microengine has eight hardwire threads. Since all the microengines can operate in parallel, assigning one packet to each thread on a microengine allows the processing of several packets in parallel. The microengines are fully programmable, general-purpose engines and can be programmed to implement any arbitrary packet processing function. Additionally, there are on-chip special-purpose hardware units for hashing and CRC compution. The distributed memory architecture of the IXP consists of support for two different types of external memories—QDR SRAM for low-latency accesses to smaller data structures used in lookup operations and DDR SDRAM for larger storage needed for packet buffers and bulk data transfers. Both the SRAM and SDRAM address spaces are shared among all the microengines and the data in these memories is accessible to each microengine thread on an equal basis. Additionally, the processor includes an on-chip SRAM that is shared among all the microengines and a small amount of local memory per microengine. The on-chip SRAM is used for providing fast access to the processing state, packet headers, or code running on the different microengines. The local memory in the microengines is meant for caching data needed by the different threads within a microengine.

1.2.2 Organization of packet processors in routers

There are several ways in which packet processors may be organized within a router. Typically they are put on line cards such that there is a processor at each router port. This organization is shown in Figure 4. At the other extreme, there is a shared pool of processors which can be used to process packets from any port. This is shown in Figure 5. There may be several variations and combinations of these two categories. For example, some ports might have such processors on their line cards, and at the same time, there might be a shared pool of processors as well to augment the processing power of ports with high demands.

Clearly, the first organization is appropriate for high forwarding requirements, where packets coming from all the ports require processing. Here, the forwarding throughput is based on the amount of parallelism and the number of non-blocking paths through the switch fabric. The second organization is

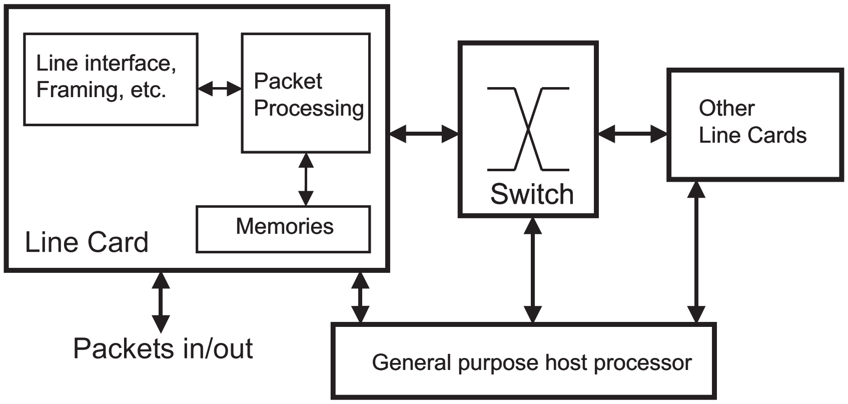

Fig. 4: A router architecture with packet processors at each port.

meaningful when the aggregate throughput requirement of the system is relatively less. In such cases, there may be one or more packet processors in the pool, which may be organized either in parallel or as a pipelined array.

In either of the two organizations, packets can be processed on the input port, the output port, or the processing might be distributed between the two ports. The advantage of the first approach is that the processing requirements are limited to the link speed of the connected link. When multicasting is used, the processing requires to be done only once. The drawback, on the other hand, is that in the case of congestion on the output link, packets that are already processed need to dropped, thereby wasting processing resources. In practice, parts of the processing is done at the input port and the remaining processing is done at the outport port. For example, to enable QoS distinction between different flows, some basic classification always needs to be done at the input. The abstract task chain shown in Figure 1, therefore, is partially implemented at the input port and partially at the output.

1.3 System-level design of embedded packet processors

1.3.1 System-level design: Issues and trends

Most of today's embedded systems are implemented as a system-on-a-chip (SoC). In the context of network packet processors, the architecture of such systems consist of a heterogeneous combination of different hardware and software components. As mentioned in the last section, the hardware components consist of CPU cores, dedicated hardware blocks, different kinds of memory modules and caches, various interconnections and I/O interfaces. All of these are integrated on a single chip and run specialized software to perform packet processing. The process of determining the optimal hardware and software architecture for such processors includes issues involving resource allocation, partitioning, and design space exploration. These are issues that are typical in most embedded systems design. To tackle the complexity of such designs, and also to meet

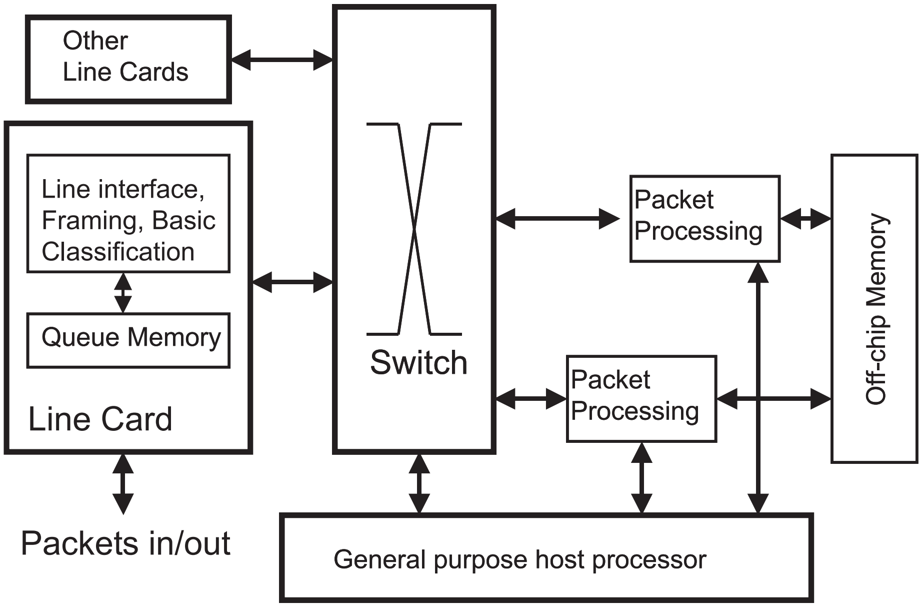

Fig. 5: A router architecture with a pool of shared packet processors.

demands for short time-to-market and low cost, several new design paradigms like platform-based design [92] have evolved.

All of these design methodologies are based on the idea of concurrent design of hardware and software, and the design flow starts with an abstract specification of the application and some performance requirements. These specifications are used to drive a system-level design space exploration [118], which iterates between modeling, verification, performance evaluation and exploration steps. Once an appropriate system architecture and hardware-software partitioning has been identified, it is followed by a hardware and software synthesis step using high-level synthesis tools. Cosimulation can then evaluate system properties like timing, memory requirements, etc., with abstract models used for the nonimplemented parts. Results from this step can be used to go through the whole process once again with a refined specification. An overview of this process, with an outline of the state-of-the-art in system-level design can be found in [57].

To speedup this entire process, increase design productivity, and reduce costs, designers today increasingly rely on the use of intellectual property (IP) blocks or “cores”, with the goal of rapidly realizing a system by assembling a network of these cores. This makes hardware design more abstract and similar to the way software is currently designed using resuable software components. Core libraries, such as the IBM Blue Logic Library [80] contain a number of verified cores that provide functions for special applications like signal processing, data compression, encryption, and several functions for networking applications. Additionally, there are also programmable processor cores. Such core libraries are supported by standard bus architectures, such as the IBM CoreConnect [81] and AMBA [7] from ARM, for interconnecting the cores.

Since the cores are predesigned and verified, a designer can concentrate on the overall system design rather than the correctness or the preformance of the individual components. This makes system-level design more effective.

The reality: In practice, the above goal in the context of SoC based embedded systems design is however far from being realised. System design is still a difficult, error prone, and time consuming process. This is mostly due to the lack of established analysis tools. Computer-aided-design tools have traditionally focussed on low-level design issues such as synthesis, gate-level timing analysis, layout and simulation. It is only during the last few years that high-level languages [140, 73], and modeling and verification frameworks like Polis [12], Ptolemy [97, 24], MOSES [108, 82, 83] and COSYMA [112] are being developed. Very recently, there has been some effort towards developing system-level design and analysis tools for core-based SoC design [21, 50].

Typically, the analysis questions faced by a designer during a system-level design process are: Which functions should be implemented in hardware and which in software (partitioning)? Which hardware components should be chosen (allocation)? How should the different functions be mapped onto the chosen hardware (binding)? Do the system-level timing properties meet the design requirements? What are the different bus utilizations and which bus or processor acts as a bottleneck? Then there are also questions related to the on-chip memory requirements, off-chip memory bandwidth, expected power consumption of the system, and layout and floorplanning issues of the design on the available chip area.

1.3.2 The case of packet processors

Although the same questions, posed above, arise in the context of designing the hardware and software for packet processors, answering them requires models and methods which do not entirely fall within the domain of traditional embedded systems design. This is because there are several characteristics specific to the packet processing domain, which do not arise in other application areas such as classical digital signal processing. In this section we point out some of these specific characteristics which necessitate looking at packet processors differently, and developing new models and analysis methods for their design.

- *New task model*: The task model underlying packet processing applications is fundamentally different from the task structure in other applications which are traditionally implemented as embedded systems. Packet processing applications process a large (or virtually infinite) stream of data items (packets). The packets from different flows are interleaved. They enter a program from an external source and are processed for a very limited time before being discarded. For each flow, there is a certain sequence of tasks which are executed, where the tasks are of high granularity and are often scheduled dynamically at runtime. This is in contrast to applications where there are recurrent or iterative computations, or where a fixed input set is manipulated with a large degree of data reuse.

- *Widely varying computation and memory requirements*: Some packet processing applications process only a limited amount of data within the protocol headers of the packet, and their processing requirements are independent of the packet's overall size. But such applications might require the maintainance of large tables or complex data structures that need to be searched or accessed on a per packet basis. Examples of this class of applications are IP forwarding, NAT, TCP connection management (for example by shaping to reduce congestion), etc.

  Other applications need computation over all the data contained in a packet and therefore require significant processing resources to process packets at line speed. Examples of this class include encryption, authentication, IP security (IPSec), data transcoding (i.e. converting multimedia data stream from one format to another within the network), etc.

- *Heterogeneous architecture*: As mentioned before, the architecture of packet processors are usually very heterogeneous and consist of a mix of different types of programmable cores and fixed-function hardware blocks implementing functions like encryption/decryption and header parsing. Therefore, issues like hardware-software partitioning, allocation, and binding of tasks to processors become more complicated and there are more design choices available compared to typical embedded system designs.

  Further, the heterogeneity makes any analysis during the system-level design phase more complicated since different models and abstractions are now necessary. For example, the different buses and processors might use different bus arbitration mechanisms and scheduling strategies, making formal timing analysis very dificult.

- *Multiple conflicting application scenarios*: In contrast to most embedded systems which are designed for a single fixed application, packet processors might be used in applications scenarios having very different requirements. For example, packet processors deployed in backbone networks can be characterized by very high throughput demands but relatively simple processing requirements per packet. On the other hand, those used in access networks have lower throughput demands but high computational requirements for each packet. The design of such processors therefore involves more complex tradeoffs than is encountered in many other application areas.

- *Large design space*: For the design space exploration of most embedded processor architectures, it is possible to formulate a parameterized *architecture template*. The design space exploration is restricted to finding appropriate values of the parameters associated with the template. These include parameters such as bus width, cache associativity, cache size, etc. The resulting design space is therefore relatively small and it is feasible to exhaustively evaluate all the possible designs by simulation. The choice of the different architectural components is also fixed (see for example [3, 128], where the system always consists of a VLIW processor, a systolic array and cache subsystem, and the design space exploration consists of identifying appropriate parameters for each of these components). However, in the context of packet processing, the heterogeneity of the processor architecture makes the design space substantially larger. It is usually not possible to identify a parameterizable template architecture. The search for such a template architecture involves a combinatorial aspect, in addition to traversing the parameter spaces of the different components. To account for this large design space, new design methodologies are required.

1.3.3 Issues in timing analysis and scheduling

One of the fundamental design challenges in the context of packet processors is to process incoming packets at line speed. Consider, for example, a line speed of 10 Gbps, with the simplifying assumption that there is no interpacket gap. Under this setup, a stream of minimum-sized packets of 64 bytes will result in a packet arrival approximately every 50 ns. Assuming a single-issue embedded RISC processor to process the packets, that executes an instruction every clock cycle and operates at 500 MHz, each instruction executes in 2 ns. This gives at most 25 instructions per packet. Considering the complexity of most packet processing functions, this is certainly not enough. To deal with this, most processors today have an array of pipelined or parallel processors to increase the allowable processing time per packet. Nevertheless, processor time is an extremely important commodity and needs to be managed itelligently.

System designers therefore need to break up the entire functionality of a packet processor into individual tasks (as described in Section 1.1.1) and evaluate possible mappings of these onto the different architectural components. For each of these mappings, by taking into account the processing and the communication times, a schedulability analysis needs to be performed to check if all the timing constraints are met.

The two main functions of a packet processor are packet processing (modifying packets) and traffic management. Intelligently managing the processing resources and the link bandwidth to support the QoS requirements of the different flows and to keep up with the line speed requires scheduling of these two resources. Further, there must be interactions between these two schedulers since, for example, dropping of already processed packets to handle link congestion results in wasting precious processing resources.

All of these above issues give rise to several problems, which we believe have not been adequately addressed in the literature, particularly due to the different characteristics of the packet processing domain as outlined in Section 1.3.2.

- Traditionally, schedulability analysis for real-time embedded systems is carried out by first modeling the system under question as some standard task model and then performing a schedulability analysis for that model. The various parameters required for the model such as different execution times and other temporal dependencies are extracted from the application using high-level design tools. In this process, the system designer is faced with two questions while choosing the model: Does the model accurately reflect all the characteristics of the application? Is the model efficiently analyzable?

We believe that most of the well known models available in the literature, such as the periodic task model [101], the sporadic task model [105], and the different multiframe models [106, 16] do not capture the essential characteristics of packet processing applications as described in Section 1.3.2. Therefore, new models are necessary which are suitably expressive and at the same time are efficiently analyzable, the two generally being contradictory concerns.

- System timing analysis in the context of packet processing architectures is a challenging problem. In traditional hardware design, system timing is usually derived by hierarchical composition of the individual component timings. This is relatively simple when component controls are single threaded and follow a fixed control sequence which only depends on input patterns. This is the usual approach in behavioral synthesis [95].

  Two main problems arise in the case of packet processors: The architecture of such systems, as already mentioned, is highly heterogeneous—the different architectural components are designed assuming different input event models and use different arbitration and resource sharing strategies. This makes any kind of compositional analysis difficult. Secondly, packet processing relies on a high degree of concurrency. Therefore, there are multiple contol threads, which additionally complicates timing analysis. Currently, the analysis of such heterogeneous systems is therefore only limited to simulation based approaches (using tools like VCC [149] and Seamless [124]). Apart from suffering high running times, these approaches also have the limitations of incomplete coverage and failure to identify corner cases. To guarantee certain timing properties within reasonable analysis times, static formal analysis based on abstract system models are required.

  The process of splitting up the entire packet processing application into a number of tasks and assigning them to different processors involves a large number of implementation choices, with each of them having a different impact on performance. Often these impacts are not entirely obvious. They can only be investigated using high-level tools to analyse the interactions between the different tasks in terms of their timing behaviour and the number of packets flowing through them, and also by analysing the utilization or load on the different processors or buses due to each of these tasks. Most of the current tools for packet processors that can enable such analysis are based on simulation and operate at a very low level, focussing on assembly instructions (an example being the environment for Intel's IXP 2xxx, called the "Transactor"). There is therefore a need for models and methods to analyse packet processing architectures and applications on a higher level of abstraction, to deduce properties like the latencies of each task (without having to delve into the low level details of its implementation) and whether all the pipeline stages of a processor are equally balanced.

- An important function of any packet processor is traffic management. Commercially available network processors either have specialized hardware to assist software with traffic management functions, or traffic management is entirely implemented in software. It may be noted that there are also a number of ASIC-based traffic management solutions available today [37].

  The aim of traffic management in packet processors is to provide QoS guarantees for a diverse range of user traffic [126]. Different traffic classes have different arrival characteristics which are specified using traffic contracts and require different QoS guarantees (provided they conform to the traffic contracts). One of the main challenges here is to devise intelligent schedulers which would support a diverse range of QoS requirements for different traffic classes. Broadly, real-time flows (voice, video, etc.) have strictly specified flow characteristics (such as maximum burst size, peak rate, long-term rate, etc.) and packets from these flows require worst case delay guarantees. On the other hand, flows arising from applications such as ftp, http, etc. do not have well specified arrival characteristics, and at the same time do not require strict delay guarantees, but may have requirements on throughput.

  There is a large body of work in the real-time systems area on integrating best-effort or soft real-time tasks into a hard real-time environment, with the goal of providing a low-delay service to best-effort tasks after satisfying the deadline constraints associated with the real-time tasks. However, all of this work pertains to the processor scheduling domain and the equivalent problem in the packet scheduling (or network traffic management) domain has remained largely ignored. In the packet scheduling area there is an extensive amount of work related to advanced buffer management and scheduling algorithms to provide QoS guarantees to real-time continuous media traffic. But relatively little has been done to exploit these algorithms to better support a wider range of traffic classes, and specifically best-effort or non-real-time traffic. In the presence of a mix of real-time and best-effort packet flows, the most widely followed scheme blindly gives higher priority to real-time packets (without considering how large are the deadlines associated with them) and best-effort packets are served only when no real-time packets are present at the scheduler. This is in spite of the fact that a major portion of Internet traffic today is still composed of non-real-time flows.

  One of the possible reasons for the lack of schedulers which support a wider range of traffic classes or which attempt to provide a better service to non-real-time flows is that the scheduling overhead for such schedulers is relatively high. We believe that with the advent of network packet processors, such schedulers can have feasible implementations and this therefore opens up this area for further research.

1.4 Thesis contributions

In this thesis we are concerned with developing models and algorithms to aid the system-level design of network packet processors. The results derived here might influence both the hardware and the software architectures of packet processors. Here we focus on system-level timing analysis and scheduling issues. More specifically, we address the three problems outlined in the last section (i.e. Section 1.3.3) and make the following main contributions:

- We identify the main characteristics of packet processing applications, and motivate the use of a new task model for the purpose of system-level schedulability analysis. This model is based on the *recurring real-time task model* which was very recently proposed in [15]. The recurring real-time task model captures the notion of conditional branches in a block of code, where the branch taken can not be determined at compile time and is known only at run time. This is an essential characteristic of packet processing applications, especially in the forwarding path, where packet classification needs to be done in several stages as we show later in this thesis. This model generalizes several well known real-time task models such as the periodic [101], sporadic [105, 17], multiframe [106], generalized multiframe [16] and recurring branching [13] models. However, the complexity of the schedulability analysis problem for this model was unknown (until recently, see [29]) and all the known algorithms had exponential complexity.

  In this thesis we show that the schedulability analysis problem for this model is NP-hard, but nevertheless can be solved in polynomial time for all practical purposes. Towards this, we introduce a new concept called "approximate schedulability analysis". It is based on the observation that if a small amount of error in the decisions made by a schedulability analysis algorithm is acceptable, then it is possible to design such algorithms to run in polynomial time and hence can be used for formal verification in any system-level design tool. We also show that this idea is fairly general and can be applied to many other task models in contexts beyond packet processor design (i.e. for general real-time embedded system designs).

  Lastly, we show that for any packet processing scenario, due to constraints on memory and also due to efficiency reasons, the number of preemptions of a task graph needs to be bounded. Towards this, for the first time, we derive the exact necessary and sufficient conditions for schedulability for this class of task models under bounded number of preemptions, and show that the concept of approximate schedulability analysis is applicable to this case as well.

  We validate our analytical results by experimental evidence, based on data from packet processing applications and using parameters from processors/architecture-blocks which are typically used to implement packet processors.

- In order to perform a system-level timing analysis of a heterogeneous packet processing architecture, where different parts of the application are mapped to different kinds of processing elements, we propose a novel analytical framework. It consists of a task model (for the application), a resource model (to model the underlying hardware i.e. the processing and the communication architecture), a model for network traffic, and a "real-time calculus" to reason about the packet flows and their processing. We show that this framework can be used to analyse arbitrarily complex and heterogeneous architectures and answer questions related to timing and also various other system properties (like the load on various processors and buses) in a single unified way. One of the main theoretical contributions of this work is that it generalizes most of the previous work on formal analysis of timing properties for heterogeneous architectures (see, for example, [122] and [121]) and shows that many results from the real-time systems area turn out to be special cases of the results that can be derived within this framework.

Since our scheme is based on abstract models of applications/algorithms and architectures, any given problem can be analysed within fractions of a second. This is in sharp contrast to simulation based approaches which require hours of simulation time. We also show that the results obtained by our formal analysis match those obtained by detailed cycle accurate simulations. By leveraging this we propose a new methodology for the design space exploration of SoC based packet processing architectures, where different stages of the exploration are based on different levels of abstractions of the architecture.

- As mentioned before, a significant portion of processing resources in any packet processor is spent on traffic management. Towards this, we propose a new scheduler to handle a mix of real-time and best-effort traffic flows, which typically passes through any router. We give theoretical guarantees that all the real-time packets would meet their deadlines and at the same time the best-effort packets would experience the shortest possible delay. Our experiments with a realistic mix of real-time and best-effort packets show that the maximum delay experienced by best-effort packets using our scheduler improve by as much as 66%, when compared to the commonly used method of serving best-effort packets only when no real-time packets are present at the scheduler. The average delay experienced by best-effort packets also improve by as much as 45%. In the case of access networks, for best-effort flows such as sporadic http requests (where response times experienced by an user are important), such improvements are clearly perceptible.

  Most of the previous work in the packet scheduling domain, to support a range of traffic classes, rely on resource reservation schemes and focus more on guaranteeing fairness. Our scheduler, on the other hand, guarantees fair sharing of the link bandwidth among the best-effort flows and also provides a low-delay service without disrupting the delay bounds associated with the real-time flows.

  From a theoretical perspective, we show that some of the well known service mechanisms from the real-time systems area, for integrating best-effort tasks into a real-time environment, turn out to be special cases of our scheme. However, the main downside of our scheduler compared to reservation based schemes is that it requires computations on a per packet basis, and can hence be computationally expensive to implement on traditional purely software based routers. We believe that application-specific packet processors can therefore be a suitable platform for implementing this and other schedulers which could not be previously deployed due to implementation overheads.

This thesis shows that the study of network packet processors, on one hand, gives rise to several interesting theoretical models and methods, and on the other hand integrates many concepts from real-time and embedded systems with ideas from computer networking, system design and computer architecture. The complex requirements of packet processors and their possibly diverse architectures provide the ideal platform for investigating methodologies for embedded systems design. We believe that all the three contributions of this thesis, as listed above, introduce concepts which apart from being interesting in their own right, also generalize previous results from system-level design, real-time systems and scheduling theory.

1.5 Related work

In this section we briefly review the state-of-the-art in research related to network packet processors. Work related to each of the three specific problems addressed in this thesis (as described in Section 1.3.3) is discussed in the respective chapters.

It is only recently that a considerable volume of research is being done on various issues related to network packet processor design. Introductory articles, containing a survey of commercially available network processors and their target application areas are [42] and [65]. [47] contains a reasonably in-depth discussion of the architectures and the associated programming environments of a number of network processors, including those from IBM, Cisco, Intel and Motorola. A comparison of the architectures of different network processors, in terms of their application domain and the position in the network where they would be used (for example, whether they would be used for low, medium, or high bandwidth communications) is done in [70]. Different architectural issues in network processors for backbone networks, related to packet processing and fast switching are discussed in [27]. Similar discussions about fundamental characteristics of network processor architectures and their implications on programability and benchmarking can be found in [117] and [8].

Architecture performance evaluation and design space exploration: There has been a number of studies related to determining the optimal architecture of network processors for various application scenarios. This includes work related to both performance evaluation and design space exploration. For example, [70] introduces a new service scheme motivated by the requirements of multi-service access networks. Based on simulation (of both hardware and software), it then evaluates different combinations of algorithms (for policing, queuing and link scheduling) along with different hardware building blocks and memory architectures, for the design of a packet processor to support the proposed service scheme. Other purely simulation based approaches, with models for, both, the application using Click [93, 94] and the hardware architecture using SimpleScalar [25] can be found in [43]. In contrast to this, SystemC [73] models of a network processor architecture, with an emphasis on the communication subsystem and the memory architecture are considered in [155]. A simulation environment for exploring multiprocessor network processing architectures that is being developed at ST Microelectronics is reported in [116].

An analytical performance model for network processors was proposed in [59, 151, 153]. Different network processor architectures can be evaluated on benchmark workloads [152] using this analytical model. We believe that this model may be classified under what can be called as a "static analytical model". Here, the computation, communication, and memory resources of a processor are all described using algebraic equations, and its performance is evaluated using traffic traces from networking benchmarks. In contrast to this class of approaches, the models presented in this thesis may be classified under "dynamic analytical models", where the dynamic behaviour of the computation and communication resources (such as the effects of different scheduling or bus arbitration schemes) are also modeled. Such dynamic models may either be based on statistical methods such as queuing theory, or may be based on theories dealing with deterministic worst case bounds. A more detailed comparison of the different performance models is done later in this thesis.

There has also been a couple of studies investigating the suitability of different classes of processor architectures (such as speculative super-scaler processors, fine-grained multithreaded processors, simultaneous multithreaded processors, and single-chip multiprocessors) for network packet processing applications [44, 46]. All of these studies are based on simulations. [45] explores the properties of different workloads in network interfaces and their execution characteristics on several high-performance processor architectures.

Programming and benchmarking: A number of papers have considered issues related to programming and benchmarking of packet processors. See [78] for an introductory article on programming of network processors. [150] gives an overview of Intel's IXP2400 network processor, focussing on its multithreaded processing and distributed memory architecture. It then describes the programming issues related to such architectures and develops a programming model for generic network applications that uses software pipelines. Based on this model, it shows how two different applications (an ATM based traffic management algorithm and an ATM AAL2-based media gateway application) can be mapped/implemented on multithreaded architectures such as the Intel's

IXP2400 processor. [129] also considers Intel's network processor (IXP1200) and reports the experiences in implementing a router using it. Here, the packet processing hardware infrastructure consisted of an Intel Pentium processor augmented with the IXP network processor. This work gives insights into how the parallelism available in a set of multiprocessors may be fully utilized when all the resources may be shared. The key technique discovered is how to statically partition the processing capacity of the microengines in the IXP for different functions, thereby avoiding the overhead involved in dynamically allocating tasks. Related to this work is [88, 89], which also describes a high-performance, distributed router implementation using general-purpose processors and network processor based programmable line cards. Similar to the work presented in [150], [5] and [75] review IBM's PowerNP network processor architecture and present the corresponding programming model. Based on this model, it is then showed how different applications related to guaranteeing QoS (by active queue management and traffic engineering), header processing (in a Genaral Packet Radio Service tunneling protocol), intelligent forwarding (using load balancing among multiple processors), payload processing, etc. can be efficiently implemented on a network processor.

Benchmarking of network packet processors has attracted considerable attention. The main complications arise out of the fact that network processors from different vendors are widely different and use varying hardware architectures and programming models. The application areas of network processors and the requirements out of them, as already mentioned before, also vary a lot—processors deployed in access networks have very different functions compared to those in backbone networks. This also makes benchmarking difficult. To address these issues most of the research has proposed either hierarchical benchmarking schemes, with each level having a different abstraction (system, function, micro and hardware) [34], or schemes which separate the different concerns like functionality, environment and the measurement methodology [147]. A couple of benchmarking suites targeted towards network processors have been used mostly for academic research. These include CommBench [152] and NetBench [104]. The Network Processing Forum (NPF) also has a Benchmarking Working Group (NPF-BWG) [10] working on various issues related to network processor benchmarking.

Scheduling and load balancing: Recently a few papers have dealt with scheduling and load balancing issues in network packet processors. In particular, the authors of [154] study a problem very similar to the one we study in Chapter 3. They consider the fact that the processor execution time associated with the processing of a packet is generally unknown and is variable. This time is dependent on the particular code that is executed, which is not known at the time the packet arrives, and also depends on the contents of the packet (in the case of payload processing applications) and the state of the processor (for example, on the cache states). This makes scheduling a difficult problem. To get around this, they hypothesize that network processing workloads tend to be predictable in nature and design scheduling algorithms based on processing time predictions. In contrast to this, we study this problem from a more theoretical perspective, and propose algorithms to handle this unpredictable nature of the code in packet processing applications. Whereas, the results in [154] are founded on experimental evidence, our algorithms have provable guarantees. In practice, it might be possible to combine these two approaches since we do not take into account the characteristics of the workload.

Lastly, [55] and [91] study a load balancing problem, where the goal is to distribute packets from a high-speed link among multiple lower-speed network processors. [55] presents a set of algorithms based on heuristics and their experiments show that the proposed load balancer reduces the associated buffer requirements to the extent that an on-chip memory would suffice. The scheme proposed in [91] is based on an adaptive deterministic mapping of the flows to the different processors. The presented simulation results show that it achieves a high processor utilization and that compared to other schemes, a larger number of router interfaces can be supported for the same processing power.

1.6 Organization and bibliographic notes

Whereas this chapter sets up the background for the work described in this thesis, the discussions in Chapter 2 are more formal and motivate the models for packet processing algorithms and architectures, and models for packet flows that are used in the later chapters.

In Chapter 3 we present the results corresponding to the first problem described in Section 1.3.3. Here we introduce a task model corresponding to packet processing applications and present several schedulability analysis algorithms for this model. The results pertaining to the complexity of schedulability analysis for this model have appeared in [29]. The conditions for schedulability under a bounded number of preemptions have appeared in [28], and the general framework for *approximate schedulability analysis* appeared in [31].

Chapter 4 addresses the second problem described in Section 1.3.3. The framework presented in this chapter is based on the theory of *real-time calculus* which was first introduced by Thiele *et al.* in [145]. The application of this theory to analyse system properties of network packet processing architectures was first shown by Thiele *et al.* in [144], and subsequently more detailed results were presented in [142] and [143]. This chapter shows that the results that can be derived within this framework generalize many results from the real-time systems area, and this work has been reported in [32]. The second part of this chapter shows that in the specific context of network packet processing architectures, the analytical results from this framework match reasonably well the results that can be obtained by detailed cycle-accurate simulations. Based on this, a methodology for the design-space exploration of packet processing architectures is also proposed. These results have appeared in [33].

Lastly, Chapter 5 contains the results corresponding to the third problem described in Section 1.3.3 and these results have appeared in [30].

2

Fundamental abstractions: Modeling algorithms, architectures and packet flows

This chapter is intended to bridge the gap between the informal discussions in Sections 1.1 and 1.2 of Chapter 1 about applications and architectures characterizing network packet processors, and the formal models used in the next three chapters.

In particular, we identify the main characteristics of packet processing applications or algorithms and their computation demands. Based on this, a task model is proposed in the next chapter, which forms the basis for our schedulability analysis algorithms. Such algorithms can then be used to determine the feasibility of a mapping of an algorithm to a given architecture. Next, we describe the two main organizations of processing elements within a packet processor—the parallel and pipelined models—and describe their relative advantages and disadvantages. This motivates our architecture model introduced in Chapter 4, which forms the basis of our analytical framework for system-level timing analysis and for determining other system properties. Lastly, we describe a traffic model for specifying packet flows, which is used both in Chapter 4 for the analytical framework, and in Chapter 5 for describing our scheduler to handle a mix of real-time and best-effort traffic classes. We conclude this chapter by briefly outlining the basic principles and properties of different scheduling disciplines which are used in the rest of the thesis. But before continuing further, we define below a few terms in the context of this thesis.

Def. 1: **(Flow)** *A flow is a sequence of packets that are treated similarly by a network node/router. This means that all packets belonging to the same flow are processed in the same way (i.e. the same processing functions are applied to them) and they have the same QoS requirements. A flow may be an aggregation of* *packets from different applications or transport layer sessions which have to be processed similarly, or depending on how packets are processed, it might also be the collection of packets belonging to the same input and output port pairs. For the purpose of this thesis, we might refer to an aggregation of different flows also as a "flow". This depends on which stage of the application a packet of the flow is in. In a later processing stage of the application, the different flows which formed the "aggregate flow" might be distinguished.*

Def. 2: **(Stream)** *We refer to a sequence of packets as a "packet stream" when we are not concerned with the flow-identifications of these packets.*

2.1 Characteristics of packet processing applications

We list below a number of typical characteristics of packet processing applications, which along with the nature of their computation demands motivate the study of new task models for representing them.

- *Computations over packet streams*: The input to any packet processing application is a virtually infinite stream of packets, each of which has to be processed within a limited amount of time, probably on multiple distributed processors, before being discarded. For each packet entering the packet processor, depending on the flow to which this packet belongs, a sequence of packet processing tasks act on it. These tasks are usually of high granularity and are often scheduled dynamically at runtime. In contrast to many other applications, there is no recurrent or iterative computation where a fixed input data set is manipulated throughout the lifetime of the computation.

- *Collection of independent tasks*: The entire packet processing functionality or computation pattern can be represented by a collection of tasks. Each task reads one or more packets from the input packet stream, either modifies them or performs some computation using them, and writes them out to be processed by the next task (example tasks, where more then one packet may be read by a task at a time, or where multiple packets might be written out by processing one packet are IP reassembly and IP fragmentation tasks respectively). The different tasks are generally self-contained and independent of each other, and therefore can be implemented on different processors with the only communication between them restricted to packet transfers. It is therefore possible to view a packet processor as a composition of the tasks with the output of each task serving as the input to another task. It will later be shown that this view enables a compositional timing analysis, where the properties of an input packet stream are modified by task and the modified properties serve as input to the next task.

- *Fixed computation pattern*: For any given flow, the set of tasks through which any packet of this flow pass is more or less fixed. Therefore, all packets of a flow are subjected to the same regular, predictable sequence of tasks. However, it may be noted that the flow to which a packet belongs is not known immediately when the packet enters the packet processor. The flow information is only known after the classification stage. Moreover, to keep up with line speeds, a packet need not be fully classified at the very begining and classification might be divided into several stages. The tasks corresponding to the different flows may be joined together to form a task graph. An example task graph processing five different flows is shown in Figure 6. Based on the results of the classification, a packet takes some path through the task graph.

Apart from Classify-tasks, there might also be a few other tasks, after crossing which a packet might take one or the other path through the task graph. An example of this in Figure 6 is the *Process IP Header* task. Depending on the outcome of the header processing, a packet might either be sent to the Classify-task or to the *IP reassembly*-task. The IP reassembly-task combines several packets to a single packet before sending it out for classification. Therefore, the path of any packet through the task graph is not known immediately when the packet enters the packet processor, but is dynamically determined as the packet progresses through the graph.

The underlying task graph for a packet processor can hence be considered to be an arbitrary directed acyclic graph, where some tasks nodes result in the control-flow to split among several other tasks nodes. Similarly, the control-flow from several tasks nodes might also join into a single task node. Given a mapping of this task graph onto an architecture, the tasks corresponding to the different nodes might execute concurrently.

- *Occasional modification of the task graph*: Although the path through the task graph for all packets belonging to the same flow is predetermined, there might be occasional changes in this path to handle special situations like packet dropping due to a congestion avoidance algorithm running for some time.

- *Out of path communications*: Apart from the packets flowing through the task graph, tasks corresponding to different nodes of the graph might communicate with each other to pass several kinds of control information. For example, a *Scheduler*-task might inform a *Queue Manager*-task (possibly implemented on a different processing element) to drop packets belonging to certain flows in response to a link congestion situation.

- *Real-time performance constraints*: To keep up with the incoming line-speed and also to satisfy the QoS requirements of certain flows, there would be real-time constraints that need to be satisfied when implementing/mapping the task graph on an architecture. Additionally, some flows might also have demands on the minimum throughput.

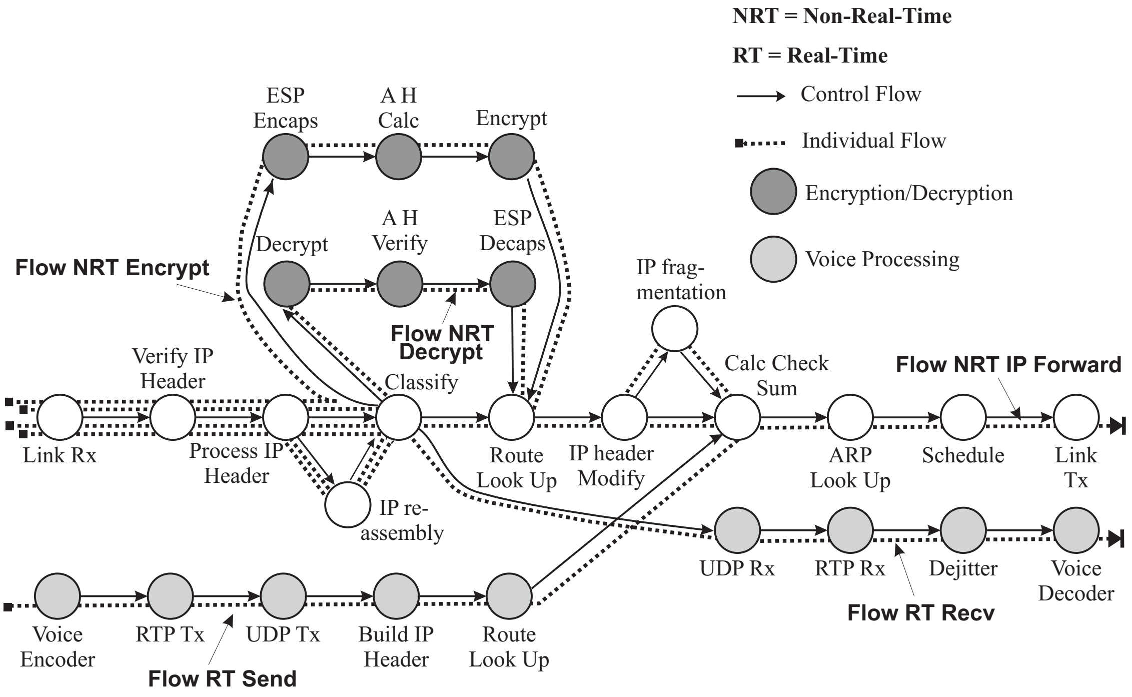

Fig. 6: A task graph for processing five different real-time and non-real-time flows. Here, there is only one *Classify* task, but in general classification might be implemented in several stages using different Classify-tasks. The solid arrows indicate the control-flow through the task graph, and the dotted lines show the path followed by packets belonging to different flows.

2.2 Organization of processing and memory units

As already mentioned in Chapter 1, packet processing architectures generally consist of multiple programmable processing elements along with dedicated coprocessors and different types of memory units. The first step towards implementing a packet processing application is to split the application into a number of tasks as described in the last section. The second step is then to map these tasks to individual processing elements or coprocessors, taking into account the performance requirement of each task—the main criteria to be satisfied is that any task mapped onto a processing element or coprocessor should complete the processing of a packet before the next packet arrives.

Multithreading: Apart from the presence of multiple processing elements, each individual element might be multithreaded, with hardware support for multithreading (for example, by automatic thread switching when the execution stalls). The commonly used practice of exploiting multithreading on a processing element is to perform the same task or set of tasks that are mapped on the processing element, in parallel on different packets. As each packet arrives, it is mapped on one thread, which executes the packet processing task on that packet. When the execution of that thread stalls, waiting for an I/O operation or memory access, another thread executes the same task on the next packet.

Let t_{min} be the amount of time (in the worst case) that can be spent in processing one packet on a processing element, before the next packet arrives at that processing element. Let f be the clock frequency with which the processing element works and I be the number of instruction cycles that must be executed corresponding to the task that is mapped onto that processing element. Then, the task fits on the processing element if $t_{min} \geq I/f$. Now assume that there are n threads on the processing element. Then, with each thread processing a packet, a thread can take upto nt_{min} time to complete the task execution on one packet. This includes both, the active execution time and the time spent in waiting for I/O or memory accesses. The active execution time is however limited to t_{min}.

As a side remark, it may be noted that although multithreading increases processor utilization, it might introduce jitter in the output packet stream emerging out of the processor. This is because of the variability in time involved in thread switching. Secondly, since a packet stays longer in a processing element in the case of multithreading, more system buffer space is required. These might be important factors to consider, at least in the case of real-time traffic.

Organization of processing elements: There are two basic organizations of the multiple processing elements within a packet processor: *parallel* and *pipelined*.

In the parallel case, the processing elements are organized in a multiprocessor configuration with each processing element executing all the packet processing tasks on a packet, from begining till completion. The different processing elements work in parallel, processing different packets (see Figure 7). The *packet dispatcher* is responsible for assigning packets, as they arrive, to the processing elements, taking into account issues like load balancing among the different processing elements and preserving packet sequence within a flow (see [91], [55] and [154] for more details). If there are m parallel processing elements in a packet processor, and as before, each processing element has n threads and the minimum interarrival time between two packets is t_{min}, then the time that can be spent in processing a packet is mnt_{min}. Of this, the permitted time for the active execution of each thread within a processing element is mt_{min}.

In the pipelined organization of processing elements, different tasks of the packet processing application are mapped to different processing elements. The processing elements are organized in a pipelined fashion, with each pipeline stage consisting of one or more processing elements. A packet passes from one stage to the next downstream stage, getting processed on the way (as shown in Figure 8). The processing elements in any pipeline stage may be optimized for specific packet processing tasks. Such processing elements are often referred to as “task-oriented processing elements”. If there are m pipeline stages with one processing element per stage, which is multithreaded with n threads, then in contrast to the parallel organization, the time that each processing element

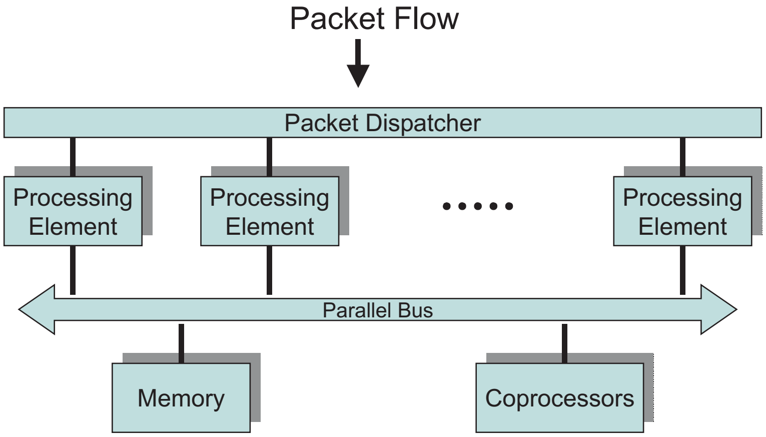

Fig. 7: Parallel organization of multiple processing elements within a packet processor with a shared memory model.

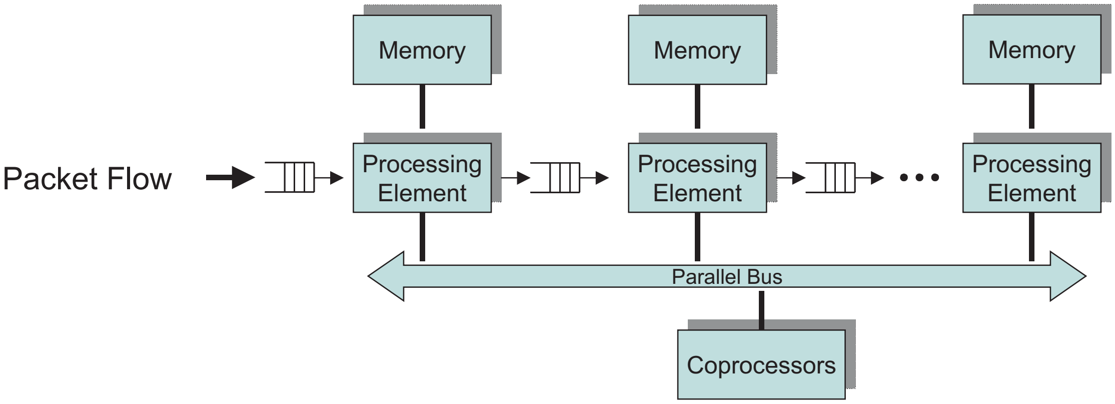

Fig. 8: Pipelined organization of multiple processing elements within a packet processor with a distributed memory model.

can spend on a packet is nt_{min}. The total processing time on a packet for the pipeline with n stages, however, as before, is mnt_{min}.

Therefore, the parallel and the pipelined models are equivalent in terms of the total processing time that can be allowed for a packet. However, in the parallel model the processing budget per packet for a single processing element is bigger and the throughput requirement is lower. But the main advantages of using a parallel model are: (a) The state of each packet can be held locally within a processing element, which allows all the packet processing functions local access to the packet state, thereby eliminating the latency of communicating this state between processing elements. (b) The different tasks of a packet processing application need not require equal time. Executing the entire application within one processor eliminates the need to break it up into several tasks, thereby avoiding questions related to partitioning, communication, etc. On the other hand, the disadvantages of the parallel model are: (a) If the entire packet processing application is too big, then the program memory space associated with a processing element might become the bottleneck. (b) The task states that are persistant across packets need to be kept in an external memory, making accesses to these states and maintaining state coherency a costly problem.

The main advantages of the pipelined model are: (a) The state of any packet processing task that is persistant across packets (for example, routing tables) can be held locally within a processing element, thereby eliminating the latency involved in accessing this state from an external memory, and also eliminating complex access/sharing mechanisms. (b) The program memory available for each task is relatively large, this being especially helpful when a task has several variations, leading to a large program memory footprint. The disadvantages of this organization are: (a) The state that is local to a packet (for example, modified packet header) must be communicated from one processing element to the next, possibly leading to a large communication overhead. (b) The mapping of the entire packet processing application onto several processing elements lead to complexities involving partitioning—each task mapped onto a processing element must be of a size that fits within a processing budget, and the different pipeline stages must also be properly balanced to maximize utilization.

Both, the parallel and the pipelined organizations therefore have their relative merits and demerits, and typically the two organizations would be combined depending on the application requirements. This would give rise to either a pipelined organization with a parallel bus, or a parallel organization with pipelined capability, depending on which requirements are more pressing for the given application.

Organization of memory units: A network packet processor typically needs three different kinds of memory. These are used for storing program or instruction code, control data and packets. Instruction memories are usually high-speed SRAMs and are organized in a way such that all the processing elements can be simultaneously fed. They can either be internal to the processor when only a small amount of code needs to be stored, or can be external in the case of code corresponding to higher-layer processing functions. Control data (such as routing tables) are stored in the control memory, which also needs to be a high-speed SRAM. For many packet processing tasks, several fields in a packet might be used for searching through tables containing routing or QoS information. Therefore, the control memory access time and bandwidth are critical factors for high-speed packet processing. Lastly, packets are stored in the packet memory. The choice of the packet memory and its organization depends, to a large extent, on the packet processing application. As an example, the requirements from a packet memory subsystem will be very different in the case of only IP-forwarding, compared to the case of TCP termination. In many cases, a packet needs to be read and written back into the packet memory several times, especially in applications involving payload processing.

There are three principal ways in which memory is organized within a packet processor: *shared*, *distributed* and *hybrid*. Although the shared memory model has the advantage that it offers a simple programming model, its main drawback

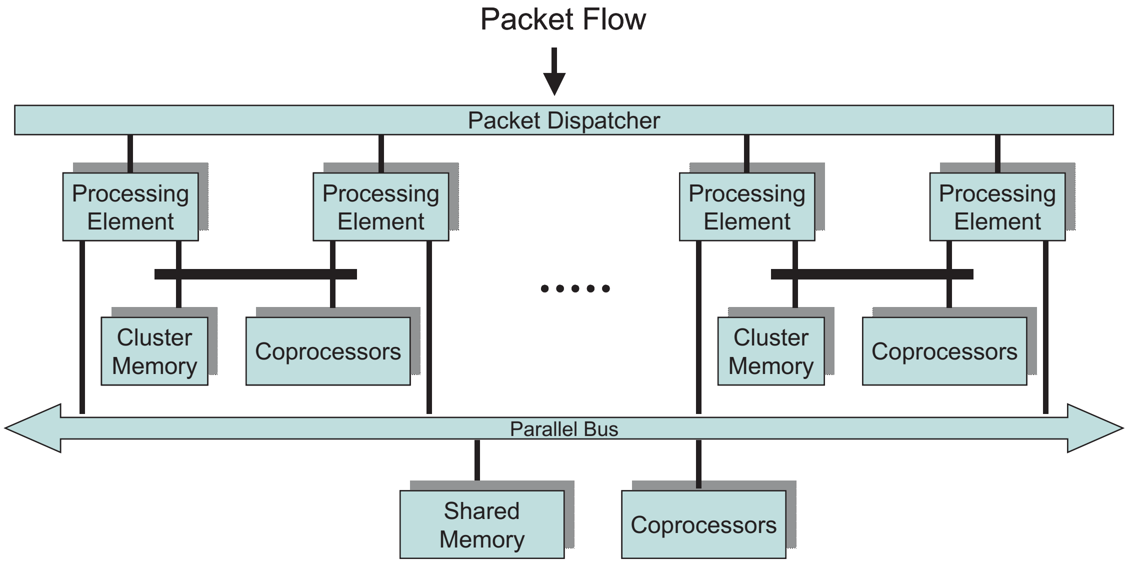

Fig. 9: A hybrid memory organization, with each cluster of processing elements having a local memory. The shared memory can be accessed by processing elements belonging to different clusters.

is that it is not scalable. Figure 7 is an example of this memory model, where a number of processing elements share a single memory unit over a shared parallel bus. An example of a distributed memory model, with each processing element having its own memory is shown in Figure 8. Although this organization is more scalable, it is also more difficult to program.

Naturally, it is possible to combine the two models to obtain a hybrid memory model. Such an organization is shown in Figure 9. Here, the processing elements are partitioned into different clusters, with each cluster having its shared cluster memory. This memory can be used to serve as an instruction and packet memory, with the instruction code replicated in all the clusters to avoid inter-cluster memory accesses. If the packet dispatcher is programmed to dispatch all packets belonging to the same flow to a single cluster, then processing elements within a cluster can also share the control memory, with at least some of the control data being stored in the cluster memory. In such cases, only control information like large routing tables need to be stored in the shared memory that can be accessed by processing elements belonging to different clusters.

2.3 Specifying bounds on packet flows

In order to provide any form of service guarantee on the processing of packets on a per-flow basis (such as the maximum allowable packet drop rate, or that packets from all real-time flows would meet their deadlines), it is required that the packet arrivals from each flow are bounded in some way. As is apparent from the preceding discussions, a simple bound on the aggregate of all flows can be obtained from the link speed and the minimum packet size. This can be specified in the form of the minimum packet interarrival time. In this section we describe means for specifying more detailed bounds on a per-flow basis. Typically, such bounds on each flow are specified as Service Level Agreements (SLAs) between a customer and a service provider. By measuring a flow's actual traffic profile, it is possible to verify whether a flow is within the specified bound. Typically, such verification is done by a *policer* after one or more classification stages. If some packets from a flow do not conform to the profile specified by the bound, then they may either be dropped or be provided a degraded service depending on the availability of resources. Additionally, packets might also be delayed to *shape* a flow according to a profile.

There are various means of providing bounds on the incoming traffic profile. They can broadly be classified as either statistical methods based on queuing theory or theories based on worst-case deterministic bounds. In this thesis, we concentrate on the latter. The deterministic bounds on traffic flows considered here are based on the application of min-max algebra to flow problems, and are widely used in the communication networks area for specifying SLAs. The theory behind this approach, now commonly referred to as "Network Calculus" was developed by Chang [35] and Cruz [48, 49] and found its final form through work by several authors. Further details can be found in two textbooks devoted to this subject ([22] and [36]). The following description mostly relies on the nomenclature used in [22].

Def. 3: **(Arrival function)** *The arrival function $a_f(t)$ of a packet flow f is defined as the number of bits or packets belonging to the flow that has arrived at a defined place in the time interval $[0, t]$. Whether $a_f(t)$ refers to the number of bits or the number of packets is either specified, or is clear from the context.*

Def. 4: **(Arrival curve)** *A packet flow f is said to be constrained by an arrival curve $\alpha_f(\Delta)$, if and only if its arrival function $a_f(t)$ satisfies the following inequality.*

$$a_f(t) - a_f(s) \leq \alpha_f(t-s) \quad \textit{for all } 0 \leq s \leq t$$

If f is constrained by the arrival curve $\alpha_f(\Delta)$, then f is also referred to as α_f-smooth.

The arrival curve $\alpha_f(\Delta)$ of a flow f can therefore be interpreted as the maximum number of bytes or packets (as the case may be), belonging to f, that can arrive within *any time interval* of length Δ. Later in this thesis, we will distinguish between *upper* and *lower* arrival curves, where an upper arrival curve is as described above and a lower arrival curve would give a lower-bound on the number of bytes or packets that can arrive from a flow within any specified length of time.

Below we give some concrete examples of specifying bounds on traffic profiles based on the concept of arrival curves.

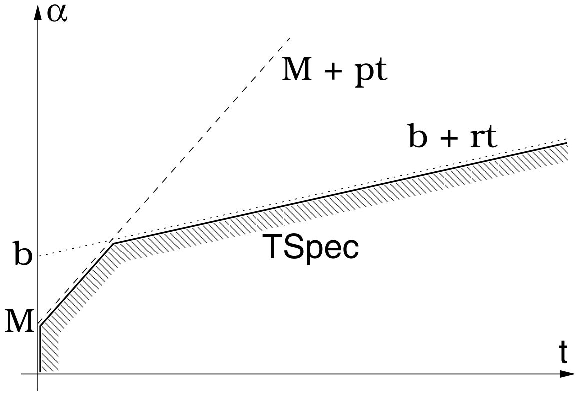

Fig. 10: Arrival curve $\alpha(t)$ of a TSpec-constrained flow.

- (σ, ρ)-*model* [48]: If the maximum burst size (i.e. the number of bits that can arrive at any instant in time) of a flow is given by σ and its long-term bounding rate is given by ρ, then the flow is said to be constrained by the arrival curve $\alpha(t) = \sigma + \rho t$.

- *TSpec* [125]: This model was introduced in the context of QoS reservations in the Internet, and can be seen as a combination of two (σ, ρ) specifications. The parameters used here to specify the maximum traffic from a flow are the *peak rate* p, the *average rate* r, *burstiness* b, and the *maximum packet size* M. A flow bounded by these parameters is given by the arrival curve $\alpha(t) = \min\{M + pt, b + rt\}$ (see Figure 10). An additional parameter m with $m \leq M$ is used to specify that packets smaller than m should be treated as packets of size m. Therefore, m determines the minimum traffic unit which needs to be policed. Note from Figure 10, that the TSpec specification can be seen as the lower envelope of two (M, p) and (b, r) specifications.

Verifying that a flow is within its specified profile: To verify that a given flow conforms to the specified SLA, *metering* is used. A typical means for metering a flow is by using a *token bucket*. A token bucket can be specified using two parameters, a capacity B and a fill rate R, and works as follows. The bucket is continuously filled at the rate R using tokens which represent units of Bytes, upto the level B. The bucket is initially filled up with tokens and traffic is allowed to pass the token bucket in the presence of a sufficient number of tokens. With each packet passing the token bucket, the number of tokens equal to the length of the packet is taken away from the bucket and the packet is marked as conforming to the profile specified by the token bucket. If sufficient number of tokens are not present when a packet arrives, then the packet is marked as non-conforming and no tokens are removed from the bucket. Therefore, the token bucket with parameters (B, R) ensures that a flow can have a burstiness of only up to size B. Since the bucket is filled at a rate R, conforming traffic is also guaranteed to be bounded in the long term by R. The conforming traffic can hence be described using the (σ, ρ)-model described above, with $\sigma = B$ and $\rho = R$. An example of a metering algorithm based on a token bucket is the ATM's generic cell rate algorithm (GCRA) [9].

Rather than having a single token bucket, it is also possible to have multiple token buckets running in parallel and a packet being marked as conforming when it conforms to all the buckets. A flow conforming to a TSpec specification can be metered using such a conjunction of two parallel token buckets—one bucket has a size M and is filled at rate p (where p is the peak rate in the TSpec), and the other bucket has size b and is filled at rate r (where r is the average rate in the TSpec). A packet passing this combination of two token buckets is marked as conforming only when there are sufficient tokens in both the buckets, and in this case tokens are removed from both the buckets. It can trivially be seen that this combination can check if a flow conforms to a TSpec with the arrival curve $\min\{M + pt, b + rt\}$.

Other combinations of token buckets include the nesting of two token buckets [77]. If the two buckets have parameters (b, r) and (B, R) with $b \leq B$ and $r \leq R$, a packet is marked as conforming to the nested combination of the buckets if there are enough tokens in both the buckets. In this case, a number of tokens corresponding to the packet size is taken out from both the buckets and the packet is marked for the highest class of service (or premium service). However, if the packet does not conform to the (b, r)-bucket, but conforms to the (B, R)-bucket, then the packet is marked for a lower quality (or degraded) service and tokens are taken out only from the (B, R)-bucket. If there are not enough tokens even in the (B, R)-bucket, then the packet is marked as non-conforming and no tokens are taken out from any of the two buckets. Such a nested combination of two token buckets can therefore be used to check if the arrival curve of a flow lies between two arrival curves specified using the (σ, ρ)-model.

Shaping a flow to conform to a specified profile: Rather than dropping packets belonging to a flow when it does not conform to a specified profile, it is also possible to hold them in a buffer so as to delay them, and then release the packet when it conforms to the profile. All the mechanisms for metering, described above, can also be used to shape a flow. For example, in the case of a single token bucket, a non-corforming packet will be held in the buffer until there are enough tokens in the bucket.

Apart from token buckets, a *leaky bucket* can also be used as a shaper. The model underlying a leaky bucket can be described as follows: Incoming network traffic is assumed to be pouring into a bucket with capacity B. The outgoing shaped traffic continuously leaks out of the bucket through a hole at the bottom (assuming a fluid model for the traffic), at a constant bit rate r, provided there is traffic (fluid) in the bucket. Packets from the incoming flow are lost if the bucket overflows. A leaky bucket can therefore tolerate a certain burstiness in the incoming traffic, but only to the extent till which the bucket does not overflow. The outgoing traffic is always generated at a constant bit rate, provided there is some backlog. In contrast to this, the output in the case of a token bucket shaper can be bursty since it allows traffic to pass whenever there are enough tokens in the bucket.

2.4 Scheduling disciplines

This thesis is concerned with both, link scheduling (in Chapter 5) and the scheduling of processing and communication resources such as buses (in Chapters 3 and 4). Nevertheless, the distinction between link and processor scheduling in the context of this thesis is relatively fuzzy, because in both the cases, a stream of packets arrive at a scheduler which manages either processing resources (which process the packets) or the link (which is used to transmit the packets). The only distinction between the two cases is the following. In the case of link scheduling, the link bandwidth required to transmit the packet is known in advance, since the packet size can immediately be determined when the packet enters the packet processor. However, this need not be the case with processor (or communication resources) scheduling. The flow-identification of a packet may be done in several stages, and hence the processor bandwidth demand for a packet can not be determined apriori when the packet enters the packet processor. After some processing is done on a packet, only then can it be determined, how much more processing needs to be done. This gives rise to some new scheduling issues which are the subject of Chapter 3.

In this section we briefly review some scheduling disciplines which are referred to throughout the rest of the thesis. As mentioned above, since we are concerned with scheduling a stream of packets belonging to different flows (be it in the context of processor or link scheduling), our setup bears more resemblance with traditional link scheduling. Hence, the description in this section is mostly done from this perspective.

- *First come first served (FCFS)*: This is possibly the simplest among all scheduling disciplines. Incoming packets from all the flows are stored in a single queue and are served in the order of their arrival. The only advantage is that it is simple to implement: insertion, deletion from the queue, and scheduling decisions, all require only $O(1)$ time. Additionally, unlike other scheduling disciplines, no per-flow state needs to be maintained. This scheduling disciplile also does not readily lend itself to providing any delay or rate guarantees. It also does not provide any isolation to the different flows, and does not have any fairness properties. However, one way to provide a delay bound is to limit the size of the queue of waiting packets. But packets need to be dropped when the queue is full. Nevertheless, because of its inherent simplicity, FCFS is one of the most widely used scheduling policies. When combined with sophisticated queue/buffer management policies which retain per-flow states, it can also provide some isolation properties [74, 139].

- *Static priority*: In this case, the different packet flows are classified into a fixed number of static priority levels. The scheduler maintains a separate queue for each priority level, and queues up packets from all flows belonging to a certain priority in the corresponding queue. A packet from a lower priority queue is served only when all the higher priority queues are empty. Each queue is served in a FCFS manner. Therefore, selecting a packet for transmission only depends on the number of priority levels and is independent of the number of flows being scheduled. Since the number of priority levels is a fixed constant, this operation takes $O(1)$ time. Similarly, queuing a packet also takes constant time.

  While this scheduling discipline does offer some form of service differentiation among the different flows—the high priority flows being protected from the lower priority flows—it does not does not allow any end-to-end performance guarantees on a per-flow basis. It only allows flows belonging to some classes to receive better service than flows belonging to some other classes. However, end-to-end delay guarantees for the different flows can be given when the flows are constrained by arrival curves of the form mentioned in Section 2.3. With all flows belonging to the same priority level having the same delay bound, but different arbitrary delay bounds being associated with the different priority levels, necessary and sufficient conditions for schedulability under static priority scheduling with the flows being constrained by concave arrival curves was derived in [48]. For schedulability conditions with flows bounded by a different form of arrival curve, see [156, 157]. The necessary and sufficient conditions for schedulability with the different flows being constrained by arbitrary arrival curves were derived in [100]. Theoretically, such delay guarantees with the flows constrained by arrival curves, should also be possible to derive in the case of FCFS scheduling. But it seems that in the case of FCFS, deriving any tight bounds (or necessary and sufficient conditions for schedulability) is substantially difficult, and this is still a topic open for research [22].

- *Fair queuing (WFQ, GPS, etc.)*: In contrast to static priority algorithms, where the priority of each flow always remains constant, dynamic priority algorithms change the priorities of the flows at run-time. The fair queuing class of algorithms fall under such dynamic priority algorithms and have many properties related to end-to-end delay guarantees and isolation among flows, which are not present in static priority based algorithms.

  Most fair queuing algorithms are based on the *Generalized Processor Sharing* (GPS) scheduler, which is a theoretical construct based on the notion of processor sharing. The basic idea of GPS was introduced in [53] under the name "fair queuing" and was analysed in [113, 114, 115]. The operation of this scheduler is defined in terms of a fluid model of the traffic and tight end-to-end delay bounds can be computed for the different flows when they traverse multiple links, each of which is served by a GPS scheduler. The scheduler works as follows. Each flow i, $i = 1, \ldots, N$, is assigned a weight (positive real number) ϕ_i, and the available link capacity at any point in time is shared among the backlogged (or active) flows in direct proportion to their weights. Packets from the different flows are queued up in different FIFO (first in first out) queues. During any time interval $(t_1, t_2]$, let $B(t_1, t_2) \in \{1, \ldots, N\}$ denote the set of flows that are continuously backlogged during this time interval. The fluid server simultaneously serves the packets from the heads of these backlogged queues, such that each backlogged queue i is served at a rate $r_i(t)$ which is given as

$$r_i(t) = \frac{\phi_i}{\sum_{j \in B(t_1,t_2)} \phi_j} R(t), \quad i \in B(t_1, t_2), \; t \in (t_1, t_2]$$

where $R(t)$ denotes the (possibly variable) link speed at time t.

This implies that each flow i ($i \in \{1, \ldots, N\}$) at any time instant t is guaranteed a minimum service rate of $\phi_i R(t) / \sum_{j=1}^{N} \phi_j$. Moreover, if some flows do not have any backlogged traffic, then the unutilized link capacity from these flows is distributed among the backlogged flows in direct proportion to their weights. The most important feature of GPS is that it handles the excess traffic from the different flows in a fair manner, irrespective of the amount of the excess traffic. If i and j are any two flows backlogged during the time interval $(t_1, t_2]$, let $W_i(t_1, t_2)$ and $W_j(t_1, t_2)$ denote the amount of traffic from i and j respectively, that are served during this time interval. Then GPS ensures that the following relation holds.

$$\frac{W_i(t_1, t_2)}{\phi_i} = \frac{W_j(t_1, t_2)}{\phi_j}$$

A bound on $|W_i(t_1, t_2)/\phi_i - W_j(t_1, t_2)/\phi_j|$ for any two flows that are continuously backlogged over the interval $(t_1, t_2]$ is often used to compare the relative fairness of different scheduling disciplines [68]. Therefore, the fairness measure of GPS using this approach is equal to zero.

As mentioned above, GPS is only a theoretical construct since packets from different flows can not be sent out simultaneously on a single link. The *Packetized Generalized Processor Sharing* (PGPS) [114], which is also referred to as the *Weighted Fair Queuing* (WFQ) [53] algorithm, uses GPS as the reference model. Here, the arrival of a packet is modeled as the arrival of a certain volume of fluid in the reference GPS system. The packet is considered to have completed transmission when the corresponding fluid volume is completely served by the GPS scheduler. The WFQ algorithm simulates the reference GPS scheduler as we just described and computes the departure times of the packets in the GPS system. At any point in time when a scheduling decision needs to be made, the algorithm selects for transmission, the packet that would have departed the earliest in the GPS system, and serves this packet. It has been shown in [114] that the packetized system, in the worst case, can only be one packet length behind the service provided by the ideal GPS system. However, WFQ suffers from a large scheduling overhead and a number of variants of this algorithm have been proposed which involve a tradeoff between the complexity of the scheduling algorithm, the offered fairness measure, and the associated delay bounds that can be derived. These include *Self-Clocked Fair Queuing* (SCFQ) [68], *Start-time Fair Queuing* (SFQ) [69], *Minimum Starting-tag Fair*

Queuing (MSFQ) [41], the *Rate-Proportional Servers* (RPS) [137] such as the *Starting Potential-based Fair Queuing* (SPFQ) and *Frame-based Fair Queuing* (FFQ) [136], and algorithms such as the *Worst-case Fair Weighted Fair Queuing* (WF^{2}Q) [19, 20].

- *Round-Robin*: This class of schedulers are intermediate in complexity compared to static priority schedulers and WFQ based scheduling algorithms. In the former class of schedulers, a high priority flow can starve a low priority flow for an indefinite amount of time. One possibility of bounding this time interval is to limit the amount of time for which any priority level can be served. After the queue belonging to a certain priority level is served for a predefined amount of time, the scheduler proceeds to the next lower priority level and serves the flows belonging to this level if there is any backlog. After having served the lowest priority level, it again begins with the queue of the highest priority level. This way, a certain share of the link bandwidth is guaranteed to all priority levels even in the presence of excess traffic from some flows. By adjusting the time quantum for which flows of a certain priority are served, this method provides a flexible, but low complexity means of sharing the link bandwidth. However, the fairness properties or the delay bounds are not as attractive as those in the case of WFQ.

  This idea behind designing schedulers was introduced in [76, 109]. Other, more flexible varieties of Round-Robin schedulers are the *Hierarchical Round-Robin* (HRR) [87] and the *Deficit Round-Robin* (DRR) [127]. Round-Robin schedulers have also been extended with ideas from the WFQ class of algorithms. DRR can be considered to fall in this class, the other algorithm being the *Virtual Time-based Round-Robin* (VTRR) [40].

- *Earliest Deadline First (EDF)*: Here a deadline is assigned to each packet as it arrives, which is the sum of the packet's arrival time and the delay guarantee associated (i.e. the maximum time the packet can spend waiting to be served) with the flow to which the packet belongs. Whenever a scheduling decision needs to be made, the scheduler selects the packet with the smallest deadline and serves it. EDF based scheduling has the property that it is optimal in terms of minimizing the maximum lateness of packets [63], where lateness is defined as the difference between the time a packet is completely served and the assigned deadline of the packet (lateness being equal to zero when this quantity is negative).

  GPS based algorithms, as mentioned above, also provide tight delay guarantees. By choosing suitable weights corresponding to the different flows, it might be possible to meet a specific set of deadlines associated with the flows. However, these weights are tightly coupled with the arrival rates of the flows. Therefore, to guarantee a small end-to-end delay to a flow, it will be necessary to assign a relatively large rate to this flow. This might lead to inefficient usage of resources and a smaller *schedulability region* [63], especially when a low bandwidth flow requires a small end-to-end delay guarantee. EDF overcomes this problem since the delay guarantees and the bandwidth requirements of a flow are decoupled in this case.

The complexity of implementing EDF is relatively high (the scheduling overhead is higher than static priority based schedulers), because at any time the scheduler needs to pick up the packet with the smallest deadline. However, since packets belonging to the same flow are always served in a FCFS order, if there are N flows with deadlines being associated with the flows, then in $O(\log N)$ time it can be determined which packet is to be served next. The schedulability condition for EDF (i.e. if all packets can meet their deadlines), when all the flows are constrained by arrival curves, has been derived in [100]. A comprehensive survey of different aspects of the EDF scheduling discipline can be found in [134].

3

Schedulability analysis

In this chapter we study the feasibility of mapping the different tasks (in the sense described in Section 2.1) of a packet processing application onto the different architectural components of a packet processor.

We consider the general setup where the processor might be connected to several input ports through which packets flow in, possibly at different line speeds, and after being processed they are put out on the appropriate output ports. The stream of packets through each input port is an aggregate of several flows, and the flow-identification of a packet is determined as it passes through the different classification tasks of the packet processing application. Corresponding to each input port is a block of code implementing the packet processing application that handles all the packets flowing through that port, and all such blocks of code execute concurrently. The packet processing application running at each input port typically depends on the source of the packets (or the constituent flows that make up the input packet stream), and its high-level task structure would be similar to the task graph shown in Figure 6. As described in Section 2.1, the vertices of such a graph represent different packet processing tasks which get triggered by incoming packets, or by preceding vertices when they complete execution and are ready to forward the packet for further processing. All the vertices of the concurrently executing task graphs for each input port are mapped onto the different processing elements (and also possibly to different communication resources, depending on whether the task graph contains "communication tasks") available on the packet processor. These processing resources, as explained in Sections 1.2 and 2.2, would typically include one or more general purpose processors, several processing engines (such as RISC cores) and special purpose hardware assists for tasks like classification, encryption, etc.

In the process of choosing a particular mapping of the vertices of the task graphs onto the different processing elements, a system designer is usually faced with several choices. One primary requirement out of any feasible mapping is that the different real-time constraints associated with the task graphs need to be satisfied. These real-time constraints can be computed from the input line speeds and the average or minimum packet sizes (as explained in Section 1.3.3), and they set an upper bound on the total time that can be spent in processing any packet. Additionally, many flows, such as those arising from voice or video processing applications, have QoS requirements which give rise to deadlines being associated with the packets.

The work in this chapter results in formal models and algorithms which can be used to quantitatively determine whether, for a givel mapping of the vertices of the task graphs onto different processing elements, all the associated real-time constraints can be satisfied or not. This question is posed as a schedulability analysis problem for a task model which captures the essential characteristics of packet processing applications. The model we study here is based on the *recurring real-time task model* recently proposed in [15], which generalizes many well known task models from the real-time systems area.

In attempting to answer the schedulability analysis question, we make the following contributions.

- We show that the schedulability analysis problem for the proposed model is intractable under both, earliest deadline first (EDF) and static priority scheduling, even for a very restricted setup. Apart from being of independent interest in the context of the problem studied in this thesis, our model forms the core of the recurring real-time task model. Our intractability result also establishes the NP-hardness of the preemptive uniprocessor version of schedulability analysis for the recurring real-time task model, under both dynamic and static priorities. Till now, no complexity results for this model were known and all the existing algorithms had exponential complexity [15].

- However, for both the scheduling disciplines, we show that it is possible to derive pseudo-polynomial time algorithms. In view of the NP-hardness result, this is the best that can be obtained in terms of exact algorithms (“exact” or “optimal” algorithms are in contrast to “approximation algorithms” [79]).

- We introduce a new concept called “approximate schedulability analysis” using which it is possible to obtain a polynomial time algorithm if a small and bounded error in the decisions made by the algorithm is acceptable. The maximum allowable error can be given as an input to the algorithm, and the smaller the value of this error, the higher is the running time of the algorithm. We also show that this concept is fairly general and can be applied to a wide variety of task models known from the real-time systems literature. All the previously known algorithms for schedulability analysis for these task models either had exponential complexity or at best could be solved in pseudo-polynomial time.

The notion of approximate schedulability analysis results in these problems being efficiently solvable for all practical purposes.

- Since due to memory constraints and efficiency reasons, the number of task preemptions in any packet processing application needs to be bounded, we derive the exact necessary and sufficient conditions for schedulability for the studied task model under a bounded number of preemptions. These conditions are substantially more complex compared to the preemptive case, and no such conditions were known either for the recurring real-time task model or for any other related model. Further, we show that the concept of approximate schedulability analysis can be used to test these conditions as well.

The rest of this chapter is structured as follows. In the next section we point out the main difficulty behind designing an algorithm for schedulability analysis for the proposed task model and why the known techniques for other task models from the real-time systems area prove to be inadequate. The task model is described in Section 3.2, which is followed by the hardness results in Section 3.3. In Section 3.4 we present the basic (exponential time) algorithms for preemptive schedulability analysis under EDF and static priority. In Section 3.5 we introduce a restricted task model to illustrate our pseudo-polynomial time algorithms, which form the basis for approximate schedulability analysis. The conditions for schedulability under a bounded number of preemptions are introduced in Section 3.6. Finally, in Section 3.7 we present the general framework for approximate schedulability analysis, and give our experimental results in Section 3.8 based on the running times of different packet processing tasks on processing elements which are typically used to implement packet processors. These results illustrate the tradeoffs between the running times of a schedulability analysis algorithm and the accuracy, for different values of the input error parameter.

3.1 Background

Starting with the periodic task model of Liu and Layland [101], a number of different models were proposed in the real-time systems literature over the last years to correctly represent real-time systems, verify timing constraints and answer scheduling-theoretic questions arising in these systems. The different models represent tradeoffs between being as general as possible (so that the characteristics of the underlying system can be accurately captured) and being efficiently analyzable. However, all of these models are based on an abstract framework in which a real-time system is modeled as a collection of independent *tasks*. Each task generates a sequence of *jobs*, each of which is characterized by a *ready-time*, an *execution requirement*, and a *deadline* (for the ease of exposition, we sometimes refer to a *job* as a *task*, when it is clear from the context which task generated the job). The sequence of jobs that can be generated is typically constrained by some rule specific to the task model, and the differences among the various models are in terms of the task structure and the rules according to which jobs are generated. The *schedulability analysis* of such a system is concerned with determining whether, for all possible sequences of jobs that can be generated, it is possible to assign to each job a processor time equal to its execution requirement, between its ready-time and its deadline.

Complexity of schedulability analysis: As an example, in the context of a real-time embedded system, a task might model an event-driven block of code running within an infinite loop, which gets triggered by external events and is required to be executed within a given deadline. The system consists of a collection of such code blocks representing different event handlers and the schedulability analysis is required to answer whether under all possible event-triggering sequences, the code blocks can be executed so that all real-time constraints are met. Such a scenario is commonly encountered during the system-level design step of embedded systems. To support this step, system-level design tools need to implement efficient algorithms for schedulability analysis for the task models supported by the tool. Unfortunately, for most realistic task models the schedulability-analysis problem is intractable (usually NP-hard) and therefore known algorithms either have exponential complexity, or at best can be solved in pseudo-polynomial time. Under these circumstances, current research in the real-time systems area has focussed either on obtaining efficient pseudo-polynomial time algorithms for these models, or on finding polynomial time algorithms for special cases of these models. Either of these solutions nevertheless largely restricts the use of such models for the design and analysis of realistic systems.

In the context of the problem studied in this thesis, i.e. system-level design of network packet processors, apart from the high complexity of the schedulability analysis algorithms for known task models, there are other problems related to modeling which result in the known task models being inadequate. This is explained below in further details.

Task structure: Recall from the discussion at the begining of this chapter, that corresponding to each input port there is a block of code responsible for processing all packets flowing in through that port. The high-level task structure for such a block of code would generally be similar to Figure 6. Let the stream of packets flowing in through a given port be composed of two flows f and f' and let the tasks to be executed on any packet belonging to flow f be t_1, t_2, t_3, t_4, t_5, and the tasks corresponding to flow f' be t'_1, t'_2, t'_3, t'_4. Typically, the case is that some of the tasks belonging to the two different flows are identical, while the others are different, i.e. say, t_1, t_2 and t_5 are identical to t'_1, t'_2 and t'_4 respectively, while t_3, t_4 and t'_3 are different. The task graph corresponding to the code which processes f and f' therefore consists of seven tasks (t_1, t_2, t_3, t_4, t'_3 and t_5), with a conditional branch after the task t_2 (where the branch taken depends on whether the packet belongs to f or f'), and the control flows from t_4 and t'_3 merge to t_5. Such conditional branches and merging of the control flow is a primary characteristic of task graphs representing control-dominated applications. This is in contrast to data-dominated applications like digital signal processing (DSP), which are characterised by long computation sections following a straight-line flow of control between relatively few control-flow boundaries or branches. Task graphs for control-dominated applications like packet processing, on the other hand, consist of mainly conditional branchs and control-flow-merges with relatively little computations in between. This can be seen in the task graph shown in Figure 6, and many other examples can also be found in [93] where task graphs of packet processing applications constructed out of Click modules have been shown. Most of the models and algorithms developed in the real-time systems area have focussed only on data-dominated applications like DSP, primarily because control-dominated applications like packet processing have emerged only very recently.

Timing constraints: In addition to the separate paths in the control flow graph that are taken by packets belonging to different flows, generally the timing constraints associated with these packets are also different. For example, because of strict QoS requirements, packets belonging to voice or video flows have hard deadlines within which they have to be processed. Packets belonging to a http flow, on the other hand, may not have any deadline requirements at all. Apparently it might seem that it is possible to assign a single *end-to-end deadline* to a task graph (which is equal to the smallest deadline associated with any flow), or assign a deadline to each *end-to-end path* in the task graph corresponding to the different flows (where a path is assigned the deadline equal to the deadline associated with the corresponding flow), and then use these deadlines for schedulability analysis. However, since the complete flow-identification of a packet need not happen until the packet passes through a number of tasks of the task graph, both these approaches are overly pessimistic and do not accurately capture the timing constraints associated with a task graph. To understand this, consider once again the example in the previous paragraph. Let t_1 and t_2 denote a header parsing and a classification task respectively. To keep up with the line speed, both these tasks should complete processing a packet within a fixed deadline from the packet arrival (i.e. before the arrival of the next packet), which therefore results in a deadline being associated with them. Now, if the flow f is related to a voice processing application and f' is an http flow with no deadline requirements, then any packet from f should be processed by t_3, t_4 and t_5 within a fixed deadline (which would generally be different from the deadlines associated with t_1 and t_2). However, the processing of packets from f' by t'_3 and t_5 need not happen within any fixed deadline. Clearly, all of these constraints can not be captured by either a single end-to-end deadline, or by two deadlines associated with the different flows. Note that this situation would not change even if there would have been a finite deadline associated with the processing of packets from f' by t'_3 and t_5. Here the deadline associated with t_1 and t_2 would be determined by, say, the line speed and the average packet size, whereas the deadlines associated with t'_3 and t_5 would be determined by other factors like the QoS requirements of f', and whether packets from f' are being buffered after being classified. These different deadlines can not be combined into one end-to-end deadline for packets from f'.

Typically, in data-dominated applications, processing starts with the receiving of a data sample and ends with sending out of a processed sample. The data stream is not constituted from an interleaved set of flows with different deadline requirements [103]. Hence, it is sufficient to have a single end-to-end deadline constraint from the start to the end of the processing and such systems can be naturally represented by known real-time task models like periodic [101], sporadic [105, 17], multiframe [106, 107] and generalized multiframe tasks [16]. However, it is now being recognized that in contrast to this, control-dominated real-time systems like packet processors have more fine-grained timing constraints [54, 56], which can not be accurately captured by the traditional real-time task models mentioned above [14]. To overcome this shortcoming, of late, new models like the recurring real-time task model [15] are being proposed. Following the same approach as in this model, for the purpose of schedulability analysis of packet processing tasks, we annotate the vertices of a task graph representing a packet processing application with execution requirements and deadlines. The schedulability analysis problem is then to determine whether all such deadlines can be met or not.

To illustrate the main difficulty involved in the schedulability analysis of a collection of task graphs whose vertices are annotated with execution requirements and deadlines, consider the following example.

```
while ($new\ packet\ arrival$) do
  execute code block $B_0$ $\{(e_0, d_0)\}$
  if ($C$) then
    execute code block $B_1$ $\{(e_1, d_1)\}$
  else
    execute code block $B_2$ $\{(e_2, d_2)\}$
  end if
end while
```

In the above code excerpt, for each code block B_i, the tuples (e_i, d_i) enclosed within the comments indicate the execution requirement and the deadline of B_i. Now, if the condition C depends on the contents of a packet or on some state of the system which can not be determined at compile time, then the *worst case branch* here would depend on the other blocks of code executing concurrently with this one. Let $e_1 = 2$, $d_1 = 2$, $e_2 = 4$ and $d_2 = 5$. If another code block is simultaneously executing with $e = 1$ and $d = 1$ then the (e_1, d_1) branch corresponds to the worst case, whereas if $e = 2$ and $d = 5$ then the (e_2, d_2) branch corresponds to the worst case. Traditionally, the schedulability analysis of code in the presence of conditional branches is done by identifying the worst case path in the code, i.e. the path which makes the most rigorous demand on the processing resources, and approximating the code by this worst case path. However, such an approach does not work in the case of our example, as shown above. The alternative, which involves enumerating all possible execution paths in the control flow graph leads to exponential complexity and is therefore computationally infeasible.

The schedulability analysis algorithms for the recurring real-time task model, as presented in [15], have a running time which is exponential in the number of vertices of the task graphs. It was remarked that this problem is likely to be intractable and in contrast to the algorithms for previous (less general) models like sporadic [105, 17], multiframe [106, 107, 16] and branching [13], the presented algorithms do not run in pseudo-polynomial time. Our results in this chapter settle the complexity of this problem by showing that schedulability analysis for this model is indeed intractable (NP-hard), but can be solved in pseudo-polynomial time.

3.2 The task model

In this section we formally introduce the task model which was motivated in the last section. This model then forms the basis for the schedulability analysis algorithms presented later in this chapter.

The code corresponding to a packet processing application is assumed to be composed of a number of packet processing tasks, whose task graph is of the form shown in Figure 6. More examples of such task graphs constructed out of different packet processing tasks can be found in [93]. We consider any such task graph T to be a directed acyclic graph with a unique *source* and a unique *sink* vertex. Each vertex of the task graph T represents the code corresponding to a packet processing task, and in case a unique source and a sink vertex do not naturally exist in a graph, then dummy vertices are added to act as source and sink vertices and these are joined by directed edges to the previous (multiple) original source and sink vertices.

Associated with each vertex v is its worst case execution requirement $e(v)$ (which can be determined prior to the run-time, i.e. say, at the compile-time of the code), and deadline $d(v)$. Whenever the vertex v is *triggered*, the code corresponding to it has to be executed (which takes $e(v)$ amount of time) within the next $d(v)$ time units. In the preemptive version of the problem, a vertex can be preempted at any point during its execution, in favour of another vertex belonging to a different graph. Whereas in the non-preemptive version of the problem, once the code corresponding to a vertex has started execution, it can not be preempted and continues executing till completion. After it completes, another vertex which has already been triggered, possibly belonging to a different task graph, can be scheduled for execution. This formulation of the non-preemptive version of the problem is motivated by the fact that once a task starts processing a packet, it might be too expensive to preempt it in favour of another packet. However, once a task has completed execution, then a different packet (possibly even from a different input port) may be processed. The tasks therefore are considered to represent the natural execution boundaries. After the processing of a packet by a task, it might be stored in the memory for some time before being processed by the next task in the task graph, possibly even on a different processing element. The alternative option of non-preemptively processing a packet by all the tasks in a task graph is overly restrictive.

Each directed edge (u, v) in the graph is associated with a minimum inter-triggering separation $p(u, v)$, denoting the minimum amount of time that must elapse before the vertex v can be triggered after the triggering of the vertex u. This can be used to model a possible communication delay between u and v, which is explained in more details later in this section.

The semantics of the execution of such a task graph state that the source vertex can be triggered at any time, and once a vertex u is triggered then the next vertex v can be triggered only if there exists a directed edge (u, v) and at least $p(u, v)$ amount of time has elapsed since the triggering of u. If there are directed edges (u, v_1) and (u, v_2) from the vertex u then only one among v_1 and v_2 can be triggered, after the triggering of u. Since the task graph is used to model a block of code which runs in a loop and processes a virtually infinite stream of packets, the triggering of the sink vertex is followed by the triggering of the source vertex, indicating the arrival of the next packet. Two such consecutive triggerings of the source vertex should however be separated by at least $P(T)$ units of time, called the *period* of the task graph.

Therefore, a sequence of vertices $v_1, v_2, \ldots, v_k$ getting triggered at time instants $t_1, t_2, \ldots, t_k$ is legal if and only if there are directed edges (v_i, v_{i+1}) and $t_{i+1} - t_i \geq p(v_i, v_{i+1})$ for $i = 1, \ldots, k-1$. The only exception is that v_{i+1} can also be the source vertex and v_i the sink vertex. In this case, if there exists some vertex v_j, $j < i$, in the sequence, such that v_j is also the source vertex then $t_{i+1} - t_j \geq P(T)$ must be additionally satisfied. The real-time constraints require that the task corresponding to vertex v_i be executed within the time interval $(t_i, t_i + d(v_i)]$.

Note that the above conditions on their own do not guarantee that a vertex u is always executed before a vertex v whenever there is a directed edge from u to v. This is because a vertex can be triggered before the deadline of the last triggered vertex has elapsed. To guarantee this property, either of the following conditions must hold: $p(u, v) \geq d(u)$, which guarantees that the vertex v can be triggered only after the task corresponding to vertex u has completed execution, or that $d(u) \leq p(u, v) + d(v)$, which guarantees that the absolute deadline of the task corresponding to vertex v is larger than or equal to the absolute deadline of the task corresponding to vertex u. In the real-time systems literature the first condition is referred to as the *frame separation property* [141] and the second as the *localized Monotonic Absolute Deadlines property* (*l-MAD*) [16]. Since the tasks to be executed on a packet should strictly be in the order specified by the task graph, throughout this chapter we assume either one of these two conditions to hold.

3.2.1 Rationale

As mentioned in Sections 1.2 and 2.2, network packet processors typically consist of multiple processing elements. The different tasks (or vertices) of the task graph are therefore mapped to different processing elements, with possibly multiple tasks mapped onto the same processor. Now consider a fragment of a task graph consisting of three vertices as shown in Figure 11(a). Let the tasks corresponding to the first and the third vertices be implemented on a processor P and the second vertex be mapped to a different processor P'. The execution requirement and the deadline for each vertex is also indicated in the figure along with the intertriggering separations. For the schedulability analysis on processor P, the subgraph to be considered is shown in Figure 11(b). If all such subgraphs (implemented on different processors) of any task graph pass the schedulability test on the concerned processors, then the overall graph is also guaranteed to be schedulable. In this context, there are two points to be noted regarding the task model. Firstly, for this scheme to work, it is necessary to associate a separate deadline with each vertex of a task graph. In Figure 11(b), it is not possible to assign a single (execution requirement, deadline)-tuple equal to $(e_1 + e_3, p_1 + p_2 + d_3)$ with the graph. This is because a continuous block of processor time equal to $e_1 + e_3$ just at the end of a time interval of length $p_1 + p_2 + d_3$ would satisfy this deadline constraint, leaving no "space" to accomodate the sequential execution of the second vertex on the processor P' in between the execution of the first and the third vertices. However, the separate deadlines assigned to each vertex as shown in the figure satisfy this requirement involved in any distributed implementation. Secondly, The intertriggering separations can be used to model a communication delay in such distributed implementations. Note that, to account for the implementation of the second vertex on a different processor P', the intertriggering separation between the first and the second vertex has been adjusted in the task graph seen by the processor P (see Figure 11(b)). If two vertices u and v, connected by a directed edge, are mapped onto the same processor, it might be sufficient to assign $p(u, v)$ to be equal to $d(u)$. To model a communication delay, in case u and v are mapped onto different processors, a larger value might be assigned to $p(u, v)$.

3.2.2 Task sets and schedulability analysis

A task set $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ consists of a collection of task graphs, the vertices of which can get triggered independently of each other. A triggering sequence for such a task set $\mathcal{T}$ is legal if and only if for every task graph T_i, the subset of vertices of the sequence belonging to T_i constitutes a legal triggering sequence for T_i. In other words, a legal triggering sequence for $\mathcal{T}$ is obtained by merging together (ordered by triggering times, with ties broken arbitrarily) legal triggering sequences of the constituting tasks graphs.

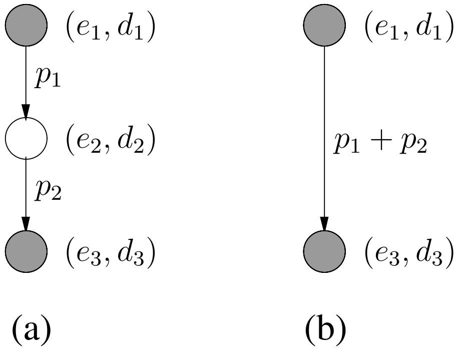

Fig. 11: (a) A task graph where vertices 1 and 3 are implemented on a processor P and vertex 2 on a different processor P', (b) The subgraph seen by processor P.

The schedulability analysis of a task set $\mathcal{T}$ is concerned with determining whether for all possible legal triggering sequences of $\mathcal{T}$, the codes corresponding to the vertices of the task graphs can be scheduled such that all their associated deadlines are met. As mentioned before, in this chapter we shall address both preemptive and non-preemptive (in the sense described above) versions of this problem. For an implementation involving multiple processing units, if the schedulability test holds for the subgraphs on the individual processors then it holds for the collection $\mathcal{T}$ of the original graphs as well.

3.2.3 Dynamic- and static-priority scheduling

Scheduling algorithms are generally implemented by assigning priorities at each time instant (according to some criteria), to all jobs (where a task, or in our context, a vertex of a task graph, when triggered generates a job) that are ready to execute and then allocating the processor to the highest priority job. Based on this, scheduling algorithms can be broadly classified into either *dynamic-priority* or *static-priority* (also known as fixed-priority) algorithms. Dynamic-priority algorithms allow the switching of priorities between tasks. This means that for two tasks, both having ready jobs at two time instants, at one instant the first task's job might have a higher priority than the second task's job, while at the other instant the priorities might switch. Static-priority algorithms, in contrast to this, do not allow such priority switching.

This classification of algorithms based on priorities leads to two different schedulability analysis questions. Given a task set $\mathcal{T}$, the dynamic-priority schedulability analysis asks whether it is possible to schedule jobs generated by any legal triggering sequence of $\mathcal{T}$, using *any* dynamic-priority scheduling algorithm such that all deadline constraints are met. The static-priority schedulability analysis question is analogous, asking the same question for *any* static-priority scheduling algorithm.

An example of a static-priority algorithm is the *rate-monotonic scheduling* algorithm [101] for a set of periodic tasks, where all tasks generate jobs periodically with their deadlines equal to the periods. The scheduling algorithm then assigns each task a priority inversely proportional to its period, i.e. the smaller the period, the higher is the priority, with ties broken arbitrarily but in some consistent manner. An example of a dynamic-priority scheduling algorithm is the *earliest deadline first* (EDF) policy. It has been proved that EDF is an *optimal* dynamic-priority preemptive scheduling algorithm in a general setup (i.e. this result is not restricted to our particular model). This means that if a set of jobs can be scheduled to meet their deadlines using any preemptive dynamic-priority algorithm, then they will also be scheduled to meet their deadlines using EDF [101]. Further, in the non-preemptive case EDF is known to be optimal for independently executing jobs if the scheduler is work conserving or non-idle (i.e. if a job is ready then it has to be scheduled if the processor is empty). Because of these results, the dynamic-priority schedulability analysis question in our case is equivalent to asking if it is possible to schedule using EDF, the jobs generated by all possible legal triggering sequences of $\mathcal{T}$. In this chapter we address both, dynamic- and static-priority schedulability analysis, for the task model described in this section.

3.3 The complexity of schedulability analysis

In this section we prove that both the dynamic- and the static-priority schedulability analysis problems for the task model described in Section 3.2 are NP-hard. We consider only the preemptive uniprocessor version of the problem, where all the tasks are mapped onto a single processor. Both the proofs show that even the very restricted case of the problem with only two task graphs where one graph consists of just a single vertex is NP-hard. Our results also establish the NP-hardness of the preemptive uniprocessor version of dynamic- and static-priority schedulability analysis for the recurring real-time task model [15]. Our proofs rely on a reduction from the knapsack problem [62] which is known to be NP-complete.

An instance of the knapsack problem is defined as follows: a finite set U, with a *size* $s(u) \in \mathbb{Z}^+$ and a *profit* $p(u) \in \mathbb{Z}^+$ for each element $u \in U$, a size constraint (or knapsack size) $C \in \mathbb{Z}^+$, and a profit goal $P \in \mathbb{Z}^+$. The decision problem is: does there exist a subset $U' \subseteq U$ such that

$$\sum_{u \in U'} s(u) \leq C \quad \text{and} \quad \sum_{u \in U'} p(u) \geq P$$

Thm. 1: *The dynamic-priority schedulability analysis problem for the task model (considered in Section 3.2) in a preemptive uniprocessor environment is NP-hard.*

Proof: Given a knapsack problem instance, we transform it in polynomial time into a task set consisting of two tasks with the property that there is a solution

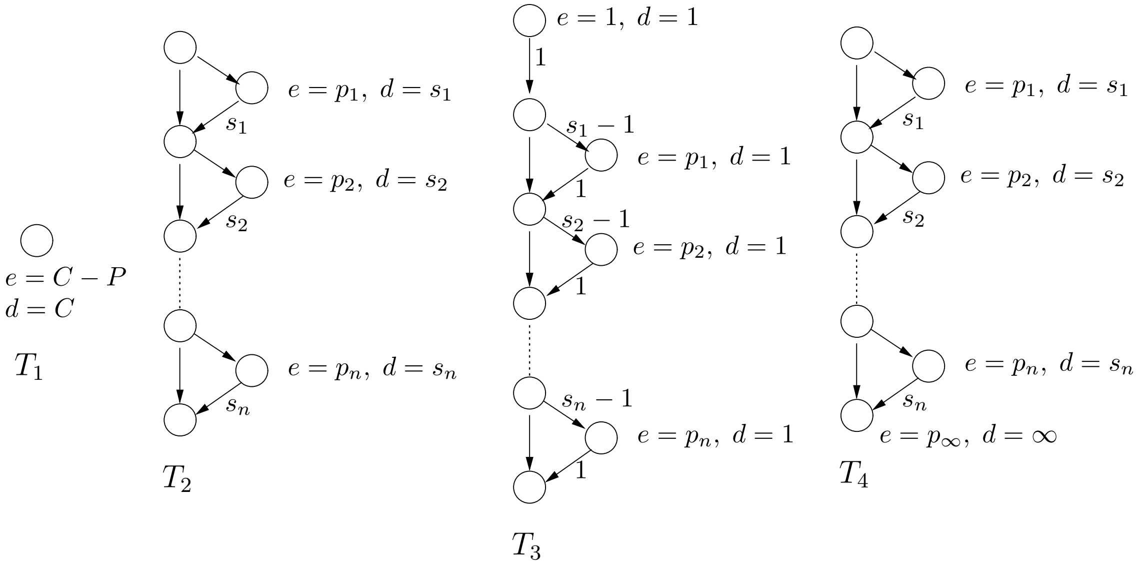

Fig. 12: Task graphs used for proving different hardness results.

to the knapsack problem achieving the specified profit goal if and only if the task set is not dynamic-priority feasible. This shows that the dynamic-priority schedulability analysis problem is NP-hard.

The knapsack problem instance specifies n items with integral sizes s_i and profits p_i, $i = 1, \ldots, n$, and an integral size constraint C and profit goal P, and asks if there exists a subset of items, the sum of whose profits is $> P$ and the sum of their sizes $\leq C$. To transform this into our problem we first scale all the profits p_i, $i = 1, \ldots, n$, and the profit goal P such that the new p_is and P satisfy the following three inequalities (this could be done, for example, by dividing all the p_is and P by $\max\{\sum_{i=1}^{n} p_i, P\} + 1$). It may be noted here that instead of scaling down the profits p_i and the profit goal P, it is also possible to construct a proof where the sizes s_i and the size constraint C are scaled up so that the constructed instance has integral e and d values.

(i) $p_i < 1,\ i = 1, \ldots, n$
(ii) $\sum_{i=1}^{n} p_i < 1$
(iii) $P < 1$

We then construct two task graphs T_1 and T_2, where T_1 consists of a single vertex with an execution requirement $e = C - P$ and a deadline $d = C$, and T_2 is as shown in Figure 12.

For each item in the knapsack problem instance, having a size s_i and profit p_i, there is a vertex in T_2 having an execution requirement $e = p_i$ and deadline $d = s_i$, and the outgoing edge from each such vertex is labelled with s_i, denoting the minimum intertriggering separation between the two vertices connected by the edge. All the unlabelled vertices and edges have their execution requirements, deadlines and intertriggering separations equal to zero. The periods of both the task graphs are equal to infinity, which means that there can be at most one run from the source to the sink vertex (both of which are the same for T_1.

The claim is that the task set $\mathcal{T} = \{T_1, T_2\}$ is not dynamic-priority schedulable if and only if there exists a subset of items from the knapsack instance, the sum of whose sizes does not exceed C and the sum of whose profits is greater than P. To see this, let us first consider the if-part. This implies that there is a valid execution of a set of vertices of T_2 having an execution requirement greater than P within a time interval of length C, if all the deadlines associated with the vertices are to be met. Within this same time interval the task T_1 can have an execution requirement equal to $C - P$. Therefore, the total execution requirement of $\mathcal{T}$ within a time interval of length C exceeds C, implying that $\mathcal{T}$ is not dynamic-priority schedulable.

For the proof in the other direction, it is to be noted that $\mathcal{T}$ will not be dynamic-priority schedulable only in the following two cases.

(i) Within a time interval of length C, T_2 has an execution requirement greater than P.
(ii) Within any time interval of length t such that

$$C + 1 \leq t = s_{i_1} + s_{i_2} + \ldots + s_{i_k}$$

where all the $i_1, \ldots, i_k$ are from $\{1, \ldots, n\}$, T_1 has an execution requirement exceeding $t - (p_{i_1} + \ldots + p_{i_k})$.

Case (ii) can not occur because within any such time interval of length t, T_1 has an execution requirement of less than C and $p_{i_1} + p_{i_2} + \ldots + p_{i_k} < 1$. Case (i) implies that there is a solution to the knapsack problem satisfying the size constraint C and having a profit greater than P. □

The following results show that the static-priority schedulability analysis problem is also NP-hard. The pseudo-polynomial time algorithm and also the approximate decision algorithms that we present later for this problem, are based on testing whether a given task from a task set is "lowest-priority schedulable". A task $T \in \mathcal{T}$ is lowest-priority schedulable if and only if all the vertices of T can always meet their deadlines with T assigned the lowest priority and all the remaining tasks of $\mathcal{T}$ having any arbitrary priority assignment. The existence of a lowest-priority schedulable task in any static-priority schedulable task set is given later in Theorem 5.

The next theorem says that testing whether a task set is lowest-priority schedulable is NP-hard, and as a corollary of this (see Cor. 1) it follows that static-priority schedulability analysis is also NP-hard.

Thm. 2: *The problem of determining whether a given task is lowest-priority schedulable is NP-hard.*

Proof: Given a knapsack problem instance as described in the proof of Theorem 1, we scale the profits and the profit goal to meet the three conditions also mentioned in that proof. Then we construct two task graphs T and T_3, where T consists of a single vertex with an execution requirement $e = C - P$ and a deadline $d = C + 1$, and T_3 is as shown in Figure 12. Here, for each item in the knapsack problem instance, having size s_i and profit p_i, there is a vertex in T_3 having an execution requirement $e = p_i$ and deadline $d = 1$, and there is one incoming edge to such a vertex labelled with a minimum intertriggering separation of $s_i - 1$ and an outgoing edge labelled with 1. The source vertex of T_3 has $e = 1$, $d = 1$, and an outgoing edge labelled with 1. As in the previous proof, all the unlabelled vertices and edges have their execution requirements, deadlines and intertriggering separations equal to zero.

We claim that the task T is not lowest-priority schedulable if and only if there is a solution to the knapsack instance satisfying the size constraint C and having a profit greater than P. If there is such a solution to the knapsack problem, then the vertices in T_3 corresponding to the items constituting the solution define an execution sequence in which T_3 within a time interval of length C can occupy the processor for an amount of time greater than P. Therefore, an execution sequence defined by the source vertex of T_3 along with these vertices, can occupy the processor for an amount of time greater than $P + 1$ within an interval of length $C + 1$, implying that T is not lowest-priority feasible.

For the other direction, it should be noted that for T_3 to occupy the processor for an amount of time greater than $P + 1$, the source vertex must always be triggered because the sum of the execution requirements of all the other vertices is less than 1. Any such execution sequence defines a solution to the knapsack instance satisfying the size constraint C and having a profit greater than P. To see this, let the sequence of vertices having non-zero execution requirements and deadlines, in such an execution sequence be the source vertex, followed by $(p_{i_1}, 1), (p_{i_2}, 1), \ldots, (p_{i_k}, 1)$ (where any tuple $(p_{i_j}, 1)$ denotes the vertex having execution requirement equal to p_{i_j} and deadline equal to 1). Then this clearly corresponds to a knapsack solution in which items having (*profit*, *size*) equal to $(p_{i_1}, s_{i_1}), (p_{i_2}, s_{i_2}), \ldots, (p_{i_k}, s_{i_k} - 1 + p_{i_k})$ satisfy the size constraint C and the profit goal P. However, since C and all the item sizes are integers, $C \geq s_{i_1} + s_{i_2} + \ldots + s_{i_k} - 1 + p_{i_k}$ implies that $C \geq s_{i_1} + s_{i_2} + \ldots + s_{i_k}$. Hence the execution sequence mentioned implies the desired solution to the knapsack problem instance. □

Cor. 1: *The static-priority schedulability analysis problem is NP-hard.*

Proof: In the proof of Theorem 2, clearly T_3 is not lowest-priority schedulable, implying that the task set $\{T, T_3\}$ is static-priority schedulable if and only if T is lowest-priority schedulable. By Theorem 2, the problem of determining whether T is lowest-priority schedulable is NP-hard. □

3.4 Basic algorithms

In this section we state the basic conditions and algorithms involved in the preemptive version of the schedulability analysis. The algorithms described here run in exponential time, but form the basis for the pseudo-polynomial time exact and polynomial time approximate decision algorithms presented in the later sections.

3.4.1 Dynamic-priority schedulability analysis

The necessary and sufficient condition for testing whether a given task set is dynamic-priority schedulable, is based on an abstraction of a task, represented by a function called the "demand-bound function". The *demand-bound function* of a task T, denoted by $T.dbf(t)$, takes as an argument a real number t and returns the maximum possible cumulative execution requirement by vertices of T that have been triggered by a legal triggering sequence and have both their ready times and deadlines within a time interval of length t. Intuitively, $T.dbf(t)$ denotes the maximum possible execution requirement that can possibly be demanded by T within *any* time interval of length t, if all its vertices are to meet their deadlines. As an example, consider the task graph T shown in Figure 13. For this graph, $T.dbf(2) = 1$ because the vertex having an execution requirement of 1 and deadline 2 can get triggered at the beginning of any time interval of length 2 and has an execution requirement of 1 if it has to meet its deadline. Similarly, $T.dbf(20) = 10$ because of a possible triggering of the two shaded vertices in the graph within any interval of length 20.

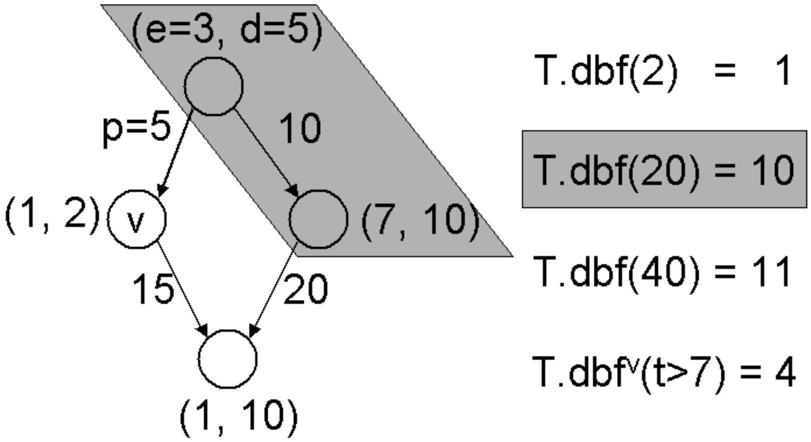

Fig. 13: The *demand-bound function* $T.dbf(t)$ for a task graph T. $T.dbf^v(t)$ denotes the demand-bound function due to triggering sequences ending at vertex v.

In addition to $T.dbf(t)$, for the purpose of non-preemptive dynamic-priority schedulability analysis (described in Section 3.6), we make use of a similar function which we denote using $T.dbf^v(t)$. For a task graph T and any vertex v belonging to this graph, $T.dbf^v(t)$ is equal to the maximum execution requirement that can be demanded by T within any time interval of length t if all the triggered vertices have to meet their deadlines, due to any triggering sequence ending at the vertex v. Therefore, for the vertex v marked in Figure 13, $T.dbf^v(2) = 1$ due to the vertex v itself, $T.dbf^v(7) = 4$ due to a possible triggering of the *source vertex* followed by the triggering of vertex v five time units after the source vertex is triggered (giving rise to an execution requirement of 4 within a time interval of length 7). Finally, for any $t > 7$, $T.dbf^v(t)$ is trivially equal to 7, because of the same sequence of triggerings as described in the case of $t = 7$.

Using the demand-bound function, the condition for dynamic-priority schedulability analysis is given by the following theorem.

Thm. 3: *A task set $\mathcal{T}$ is dynamic-priority schedulable in a preemptive environment if and only if for all $t \geq 0$, $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$.*

Proof: First we prove that the condition is necessary (i.e. the 'only if' part). This states that the task set $\mathcal{T}$ being dynamic-priority schedulable implies that for all $t \geq 0, \sum_{T \in \mathcal{T}} T.dbf(t) \leq t$, which is equivalent to the proposition that if there exists some $t \geq 0$ for which $\sum_{T \in \mathcal{T}} T.dbf(t) > t$ then the task set $\mathcal{T}$ is not dynamic-priority feasible. We will prove the second proposition.

Let for some $t > 0, T_1, T_2, \ldots, T_k$ be all tasks of $\mathcal{T}$ for which $T_i.dbf(t) > 0$ for all $i = 1, 2, \ldots, k$. At time zero, let the triggering sequence of the vertices of the task graphs corresponding to each of these tasks, which result in the computation of $dbf(t)$, be started. Then for all the vertices occurring in this triggering sequence, to meet their associated deadlines, an amount of processor time equal to $\sum_{i=1}^{k} T_i.dbf(t)$ has to be allocated within time t. This will not be possible if $\sum_{i=1}^{k} T_i.dbf(t) > t$.

Now we give the proof for sufficiency i.e. for all $t \geq 0, \sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ implies that $\mathcal{T}$ is dynamic-priority schedulable. To prove this let us assume that for all $t \geq 0, \sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ and still $\mathcal{T}$ is not dynamic-priority schedulable. If this is the case then let in some execution sequence, a vertex of $T \in \mathcal{T}$ be the first vertex that misses its deadline, when the triggered vertices of the tasks of $\mathcal{T}$ are scheduled according to the EDF policy. Recall from the discussion in Section 3.2.3 that EDF is known to be an optimal scheduling policy, in the sense that if there is any feasible schedule then EDF will also schedule the vertices such that all deadlines are met. Assume that the vertex which missed its deadline was generated at time t_r and its deadline was at time t_d. Clearly, during the entire time interval $[t_r, t_d]$ the processor was always occupied, because otherwise this vertex could have been executed during such a free time. Now let us look back in time before t_r and stop immediately before the instant at which either the processor was idle or we encounter some vertex being executed whose deadline is beyond t_d. Let us call the time instant at which we stopped as t'_r.

Clearly t'_r is a time at which some vertex of a task graph was just triggered, and all the vertices that were executed during the time interval $[t'_r, t_d]$ were triggered on or after t'_r and had their deadline before or at t_d (by our definition of t'_r). Let these vertices belong to the tasks $T_1, T_2, \ldots, T_k$ and T of $\mathcal{T}$. The processor was always occupied during the time interval $[t'_r, t_d]$, which implies that the summation of the execution requirements of $T_1, T_2, \ldots, T_k$ and T during this interval exceed $(t_d - t'_r)$. Hence, $\sum_{i=1}^{k} T_i.dbf(t_d - t'_r) + T.dbf(t_d - t_r) > (t_d - t'_r)$ contradicting our assumption that for all $t \geq 0$, $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$. □

The next theorem shows that the problem of computing $T.dbf(t)$ for a task T is NP-hard. The proof of this theorem is based on an argument similar to what was used to prove Theorem 1.

Thm. 4: *The problem of computing $T.dbf(t)$ is NP-hard.*

Proof: Given a knapsack problem instance consisting of n items, a profit goal P and a size constraint C, as described in the proof of Theorem 1, construct the task graph T_2 shown in Figure 12 and described in the same proof. Clearly, $T_2.dbf(C) > P$ if and only if there is a solution to the knapsack problem satisfying the size constraint C and having a profit greater than P. □

It may therefore be noted that any algorithm for dynamic-priority schedulability analysis based on the test given in Theorem 3 is faced with two problems which are computationally difficult. Firstly, the problem of computing the demand-bound function $T.dbf(t)$ for any t involves exponential complexity, and secondly the test involves a universal quantification over an unbounded set, and is hence not practical.

Recall from Section 3.2 that for any task graph T, $P(T)$ denotes the period of this graph, i.e. the minimum time interval between two consecutive triggerings of the source vertex of T. Let $E(T)$ denote the maximum cumulative execution requirement arising from a sequence of vertices on any path from the source to the sink vertex of the task graph T, i.e. if a sequence of vertices $v_1, \ldots, v_k$ of T be such that v_1 is the source vertex, v_k is the sink vertex, and there are no other source or sink vertices in $v_2, \ldots, v_{k-1}$ and there is a directed edge from v_i to v_{i+1}, $1 \leq i \leq k-1$, then $E(T)$ denotes the maximum of $e(v_1) + \ldots + e(v_k)$ from among all such vertex sequences.

It may now be observed that for any t, $T.dbf(t) \leq 2E(T) + tE(T)/P(T)$. To understand this, note that any legal triggering sequence of vertices $v_i, \ldots, v_j$ of a task graph T can be considered to be composed of three subsequences: $v_j, \ldots, v_{sink}$ followed by $v_{source}, \ldots, v_{sink}$ and finally $v_{source}, \ldots, v_j$, where v_{source} is the source vertex of T and v_{sink} is the sink vertex of T. If this sequence of vertices are triggered such that their triggering time and deadline fall within a time interval of length t, then the execution demand within this time interval arising out of the subsequence $v_{source}, \ldots, v_{sink}$ is bounded by $tE(T)/P(T)$. The execution demand within this time interval arising out of each of the two subsequences $v_j, \ldots, v_{sink}$ and $v_{source}, \ldots, v_j$ is bounded by $E(T)$. Hence, $T.dbf(t) \leq 2E(T) + tE(T)/P(T)$.

Using this inequality in conjunction with Theorem 3, we obtain that if a task set $\mathcal{T}$ is not dynamic-priority schedulable, then there exists some $t > 0$ such that $\sum_{T \in \mathcal{T}} T.dbf(t) > t$, which is equivalent to $t < \frac{\sum_{T \in \mathcal{T}} 2E(T)}{1 - \sum_{T \in \mathcal{T}} \frac{E(T)}{P(T)}}$. Using this inequality, we obtain the following corollary of Theorem 3.

Cor. 2: *A task set $\mathcal{T}$ is dynamic-priority schedulable in a preemptive environment if and only if for all $t < \frac{\sum_{T\in\mathcal{T}} 2E(T)}{1-\sum_{T\in\mathcal{T}} \frac{E(T)}{P(T)}}$, $\sum_{T\in\mathcal{T}} T.dbf(t) \leq t$.*

Note that it may be assumed that the quantity $\sum_{T\in\mathcal{T}} \frac{E(T)}{P(T)}$ is *a priori* bounded by a constant strictly less than one, because otherwise, the task set $\mathcal{T}$ is anyway not schedulable. $\frac{E(T)}{P(T)}$ for any task T can be referred to as its *utilization*. Therefore, using this corollary, although the number of tests to check if the sum of the demand-bound functions for any value of t exceed t, is bounded, it is pseudo-polynomial in the size of the input specification. This scheme for bounding the number of tests in a schedulability analysis algorithm is fairly standard, and has been applied to other task models as well (see also [15] and [16]).

3.4.2 Static-priority schedulability analysis

The conditions for static-priority schedulability analysis that we present in this section are based on those derived in [15] for the recurring real-time task model. In Section 3.4.4 we present a new condition for schedulability analysis which is tighter than those presented in [15] and which then forms the basis for the pseudo-polynomial and approximate decision algorithms in the later sections. In contrast to the conditions presented in Section 3.4.1 for dynamic-priority schedulability analysis, the conditions present here are only *sufficient* and not *necessary*.

As defined in Section 3.2.3, the static-priority schedulability analysis of a task set $\mathcal{T}$ is concerned with determining whether there exists an assignment of priorities to the tasks of $\mathcal{T}$ under which they can be scheduled by a static-priority run time scheduler so that all deadlines are met even in the worst case triggering sequence. Any such priority assignment is defined to be a *good* static-priority assignment for $\mathcal{T}$. As mentioned in Section 3.3, solving this schedulability analysis problem is based on testing whether a given task $T \in \mathcal{T}$ is lowest-priority schedulable. Clearly, if there is a procedure for testing lowest-priority schedulability, and the task set $\mathcal{T}$ is static-priority schedulable, then $|\mathcal{T}|$ calls to this procedure will be sufficient to identify a lowest-priority schedulable task of $\mathcal{T}$. Therefore, if $|\mathcal{T}| = n$ then with $O(n^2)$ calls to this procedure a good static-priority assignment for $\mathcal{T}$ can be determined based on the following theorem.

Thm. 5: **(Audsley, Tindell, Burns [11])** *Suppose a task $T \in \mathcal{T}$ is lowest-priority schedulable. Then there is a good static-priority assignment for $\mathcal{T}$ if and only if there is a good static-priority assignment for $\mathcal{T}\backslash\{T\}$.*

An algorithm for static-priority schedulability analysis therefore reduces to devising an algorithm for lowest-priority schedulability testing. An algorithm implementing a sufficient condition for lowest-priority schedulability was given in [15] for the recurring real-time task model, which is also applicable to the task model we consider in Section 3.2. It is also based on an abstraction of a task, similar to the demand-bound function presented in Section 3.4.1, and uses a function called the “request-bound function”. The *request-bound function* of a task T, denoted by $T.rbf(t)$, takes as an argument a real number t and returns the maximum possible cumulative execution requirement by vertices of T that have been triggered according to some legal triggering sequence and have their ready times within any time interval of length t. Intuitively, $T.rbf(t)$ is an upper bound on the maximum amount of time, within any time interval of length t, for which T can deny the processor to all lower-priority tasks. Based on this function, the following sufficiency condition was given for lowest-priority schedulability testing in [15].

Thm. 6: **(Baruah [15])** *A task $T \in \mathcal{T}$ is lowest-priority schedulable if $\forall t : \exists t' \leq t$ such that $(t' - \sum_{T' \in \mathcal{T} \setminus \{T\}} T'.rbf(t')) \geq T.dbf(t)$.*

Now, based on similar arguments that we used to prove that computing $T.dbf(t)$ is NP-hard, we next show that computing the request-bound function $T.rbf(t)$ for any t is also NP-hard.

Thm. 7: *The problem of computing $T.rbf(t)$ is NP-hard.*

Proof: Consider a knapsack problem instance as described in the proof of Theorem 1 with an additional item having a very large profit (for example greater than the sum of the profits of all the other items), which we denote by p_∞, and zero size. Now construct the task graph T_4 as shown in Figure 12. This is similar to the task graph T_1 in the same figure with the only difference being that the last vertex is labelled with an execution requirement of $e = p_\infty$ and deadline $d = \infty$. It may be noted that any solution to the knapsack problem will include this item with profit p_∞. The rest of the proof is based on similar arguments as given in the proof of Theorem 4. □

Lastly, based on exactly similar arguments as used in Section 3.4.1 for bounding the value of $T.dbf(t)$ for any t, it is possible to show that for any t, $T.rbf(t) \leq 3E(T) + tE(T)/P(T)$. Using this in conjunction with Theorem 6 and the upper bound on $T.dbf(t)$, the following can be obtained as a corollary of Theorem 6.

Cor. 3: *A task $T \in \mathcal{T}$ is lowest-priority schedulable if $\forall t < \frac{3\sum_{T' \in \mathcal{T}} E(T')}{1 - \sum_{T' \in \mathcal{T}} \frac{E(T')}{P(T')}} : \exists t' \leq t$ such that $(t' - \sum_{T' \in \mathcal{T} \setminus \{T\}} T'.rbf(t')) \geq T.dbf(t)$.*

As in Section 3.4.1, assuming that $\sum_{T' \in \mathcal{T}} \frac{E(T')}{P(T')}$ is *a priori* bounded by a constant strictly less than one, this corollary bounds the number of tests for lowest-priority schedulability to be pseudo-polynomial in the size of the input specification. Nevertheless, the algorithm for static-priority schedulability using this scheme is still of exponential complexity because of the computation of $T.rbf(t)$. Our results in the following sections show how to get around this problem.

3.4.3 Computing the *demand-* and *request-bound functions*

The last two sections introduced two abstractions of any task T—the *demand-bound function* $T.dbf(t)$ and the *resource-bound function* $T.rbf(t)$—and based on them gave the conditions for dynamic- and static-priority schedulability of a task set. In this section we give the formal algorithms for computing these two functions. Both the algorithms are of exponential complexity, but some of the techniques introduced here will be used in the later sections while deriving pseudo-polynomial time algorithms for computing these functions.

Given a task graph T, let $v_1, v_2, \ldots, v_k$ getting triggered at time instants $t_1, t_2, \ldots, t_k$, be a legal triggering sequence in the sense described at the very begining of Section 3.2. Now consider some time length $t \geq t_k + d(v_k) - t_1$. Clearly, the triggering of $v_1, v_2, \ldots, v_k$ generates a sequence of jobs whose ready-times and deadlines fall within a time interval of length t. Further assume that from all possible legal triggering sequences of the vertices of T whose resulting jobs have their ready times and deadlines within a time interval of length t, $v_1, \ldots, v_k$ getting triggered at $t_1, \ldots, t_k$ is the sequence for which the sum of the execution requirements of the vertices (i.e. $e(v_1) + \ldots + e(v_k)$) is maximized. Then by definition, $T.dbf(t) = e(v_1) + \ldots + e(v_k)$. $T.rbf(t)$ is defined exactly in the same manner, with the only difference being that instead of $t \geq t_k + d(v_k) - t_1$, we require the condition $t \geq t_k - t_1$ to hold. This is because here only the ready times of the jobs must lie within a time interval of length t. To compute $T.dbf(t)$ and $T.rbf(t)$, two different procedures are used depending on whether t is small or large. We describe below what *small* and *large* exactly means in this context and then describe how the functions are computed.

3.4.3.1 $T.dbf(t)$ and $T.rbf(t)$ for *small* values of t

These are values of t, for which the sequence of vertices that contribute towards computing $dbf(t)$ or $rbf(t)$, contain the source vertex at most once. In other words, if $T.dbf(t)$ is computed as a result of the triggering of a sequence of vertices $v_1, v_2, \ldots, v_k$ of T, then either the source vertex never occurs, or occurs exactly once in the multiset $\{v_1, v_2, \ldots, v_k\}$. Note that there may be vertices other than the source vertex which occur twice (can not be more, because that would necessitate the source vertex getting triggered more then once). To obtain such sequence of vertices, we construct a new task graph by taking two copies of the task graph T and adding an edge from the sink vertex of the first graph to the source vertex of the second graph and finally replacing the source vertex of the first graph with a *dummy* vertex with execution requirement and deadline both equal to zero. The intertriggering separations on all edges outgoing from this source vertex is also made equal to zero. The execution requirements and deadlines associated with all the other vertices, and the intertriggering separation times associated with the other edges are retained from T, and the new edge added from the sink to the source vertex is labelled with an intertriggering separation equal to the deadline of the sink vertex.

Now note that corresponding to any path in this new graph, there is a triggering sequence of the vertices in T in which the (original) source vertex of T is triggered at most once, and vice versa. If $v_1, v_2, \ldots, v_k$ be the sequence of vertices corresponding to some path π, then let $t_\pi = \sum_{i=1}^{k-1} p(v_i, v_{i+1})$ and $e_\pi = \sum_{i=1}^{k} e(v_i)$. Then clearly e_π is a lower bound on both $T.dbf(t_\pi + d(v_k))$ and $T.rbf(t_\pi)$. The path π is a witness to this. Now we enumerate all possible paths in the new graph and corresponding to each such path π, we compute t_π and e_π. Then for any small value of t (in the sense described above),

$$T.dbf(t) = \max_\pi \{e_\pi : t_\pi + e(v_k) \leq t\} \quad \text{and} \quad T.rbf(t) = \max_\pi \{e_\pi : t_\pi \leq t\}$$

where the $\max$ operation is over all possible paths π in the new graph and v_k denotes the last vertex in any such path. We point out here that since in general there can be an exponential number (in the number of vertices of T) of paths in such a graph, the worst case running time of this procedure has an exponential complexity.

3.4.3.2 $T.dbf(t)$ and $T.rbf(t)$ for *large* values of t

Here we consider values of t for which the sequence of vertices that result in the computation of $dbf(t)$ or $rbf(t)$ contain the source vertex two or more times. Given such a t, let the triggering sequence that results in the computation of $T.dbf(t)$ be $v_1, v_2, \ldots, v_k$ getting triggered at time instants $t_1, t_1 + p(v_1, v_2), \ldots, t_1 + \sum_{i=1}^{k-1} p(v_i, v_{i-1})$. Let $t_k = t_1 + \sum_{i=1}^{k-1} p(v_i, v_{i+1})$. Note that here the set $\{v_1, \ldots, v_k\}$ is a multiset with some vertices being present more than once. Now within the time interval $[t_1, t_k]$, let t_f and t_l respectively denote the time instants at which the source vertex was triggered for the first and the last time, and let these be denoted by v_f and v_l. The execution requirement of the jobs generated as a result of the triggering of $v_1, \ldots, v_k$ can now be represented as a sum of three different terms: execution requirement of jobs generated by

$$(v_1, \ldots, v_{f-1}) + (v_f, \ldots, v_{l-1}) + (v_l, \ldots, v_k) \tag{3.1}$$

As before, $E(T)$ denotes the maximum cumulative execution requirement among all possible paths from the source to the sink vertex of the task graph T, and let us denote this path by $\pi = u_1, \ldots, u_n$. Let $t_\pi = \sum_{i=1}^{n-1} p(u_i, u_{i+1}) + d(u_n)$. Now assuming that the period of the task graph $P(T) \geq t_\pi$ (which follows from both, the frame separation property and the localized Monotonic Absolute Deadlines property described in Section 3.2), the duration of the time interval $[t_f, t_l)$ is clearly a multiple of $P(T)$. Hence the maximum possible execution requirement of T over this time is $(t_l - t_f)E(T)/P(T)$, which is also equal to the execution requirement of the jobs generated by $v_f, \ldots, v_{l-1}$.

Next, consider the triggering sequence of the vertices of T in which $v_1, \ldots, v_{f-1}, v_l, v_{l+1}, \ldots, v_k$ are triggered at time instants $t_1, \ldots, t_{f-1}, t_l - (t_l - t_f), t_{l+1} - (t_l - t_f), \ldots, t_k - (t_l - t_f)$. The execution requirement of the jobs resulting out of this triggering is the same as that generated by the first and the third terms of Expression (3.1), and is exactly equal to $T.dbf(t - (t_l - t_f))$.

Note that this sequence of vertices contain the source vertex at most once and hence the interval of length $t - (t_l - t_f)$ is *small*. $T.dbf(t - (t_l - t_f))$ can therefore be computed by the procedure for computing $dbf(t)$ for small values of t. This gives us a procedure for computing $T.dbf(t)$ for any general value of t, provided we know the value of $(t_l - t_f)$. Since we know that the sequence of vertices that result in computing $T.dbf(t - (t_l - t_f))$ contain the source vertex at most once, $t - (t_l - t_f)$ must be strictly less than $2P(T)$. Further, $t_l - t_f$ is an integral multiple of $P(T)$. These together imply that $t - (t_l - t_f))$ is either equal to $t \bmod P(T)$ or $P(T) + t \bmod P(T)$. Hence,

$$\begin{aligned} T.dbf(t) &= \max\{\lfloor t/P(T)\rfloor E(T) + T.dbf(t \bmod P(T)), \\ &\quad (\lfloor t/P(T)\rfloor - 1)E(T) + T.dbf(P(T) + t \bmod P(T))\} \qquad (3.2) \end{aligned}$$

$$\begin{aligned} T.rbf(t) &= \max\{\lfloor t/P(T)\rfloor E(T) + T.rbf(t \bmod P(T)), \\ &\quad (\lfloor t/P(T)\rfloor - 1)E(T) + T.rbf(P(T) + t \bmod P(T))\} \end{aligned}$$

Hence, the functions $T.dbf(t)$ and $T.rbf(t)$ can be computed for *any* value of t, using the procedure in Section 3.4.3.1 for computing these functions for *small* values of t, without iterating over the task graph T multiple times.

3.4.4 Improved static-priority schedulability analysis

The conditions for static-priority schedulability analysis presented in Section 3.4.2 are based on the resource-bound function $T.rbf(t)$ as described in [15] in the context of the recurring real-time task model. In this section we give a modified definition of this function and show that it leads to a new sufficiency condition for testing lowest-priority schedulability which is tighter than the test given by Theorem 6 in the following sense. For any task set $\mathcal{T}$, if a task $T \in \mathcal{T}$ is returned as lowest-priority schedulable by the test in Theorem 6 then it is also returned as lowest-priority schedulable by our test, and there exist task sets $\mathcal{T}$ and tasks $T \in \mathcal{T}$ which although being lowest-priority schedulable, fail the test in Theorem 6 but are returned as lowest-priority schedulable by our test. Lastly, we show that for any task set consisting of exactly two tasks, our test is both a necessary and sufficient condition.

Our modified definition of the resource-bound function, which we denote by $T.rbf'(t)$ is similar to $T.rbf(t)$ and returns the maximum possible cumulative execution requirement by vertices of T within any time interval of length t, that have been triggered by a legal triggering sequence. To illustrate the difference between the two functions, consider a task graph T consisting of a single vertex having an execution requirement of 5 and any arbitrary deadline. Whereas $T.rbf(t) = 5$ for any $t \geq 0$ (since the ready time of T is at time 0), $T.rbf'(t) = t$ for $t \leq 5$ and is equal to 5 for any $t > 5$. Our new sufficiency condition for lowest-priority schedulability is based on the lemma and the theorem given below.

Lem. 1: *Let $T \in \mathcal{T}$ and the task graph corresponding to T have n vertices $v_1, \ldots, v_n$. If each of these vertices v_i is lowest-priority schedulable in the task set $\mathcal{T}\backslash\{T\} \bigcup\{v_i\}$, then T is also lowest-priority schedulable.*

Proof: If the vertex v_i has an execution requirement of $e(v_i)$ and deadline equal to $d(v_i)$ and is lowest-priority schedulable in the task set $\mathcal{T}\backslash\{T\}\bigcup\{v_i\}$ then within any time interval of length $d(v_i)$, tasks of $\mathcal{T}\backslash\{T\}$ never occupy the processor for an amount of time exceeding $d(v_i) - e(v_i)$. Clearly, if this is true for all vertices of T then no matter how they are triggered, they can always be executed to meet their deadlines, implying lowest-priority feasibility of T. It is trivial to see that the lemma is true even for the other direction. □

Thm. 8: *A task $T \in \mathcal{T}$ is lowest-priority schedulable if for all vertices v belonging to the task graph of T, $\exists 0 \leq t \leq d(v)$ for which $t - \sum_{T' \in \mathcal{T}\backslash\{T\}} T'.rbf'(t) \geq e(v)$.*

Proof: To prove the sufficiency of this test, we are required to prove that if $\exists 0 \leq t \leq d(v)$ for which $t - \sum_{T' \in \mathcal{T}\backslash\{T\}} T'.rbf'(t) \geq e(v)$ for all vertices v of T then T is lowest-priority schedulable. Assume that $t - \sum_{T' \in \mathcal{T}\backslash\{T\}} T'.rbf'(t) \geq e(v)$ and still the vertex v is not lowest-priority schedulable. Then let v miss its deadline at time t_d when being scheduled by a static-priority scheduler and let it be the case that v was triggered at time t_r. Therefore, within the time duration $t_d - t_r = d(v)$, the processor was occupied for more than $d(v) - e(v)$ amount of time by the higher priority tasks, contradicting the assumption that $t - \sum_{T' \in \mathcal{T}\backslash\{T\}} T'.rbf'(t) \geq e(v)$. The proof follows from the above argument along with Lemma 1. □

It is easy to see that if a task $T \in \mathcal{T}$ is returned as lowest-priority schedulable by the test given by Theorem 6 then it also passes the test of Theorem 8. Additionally, if T is returned as lowest-priority schedulable, then it is really so. To show that this represents a tigher test, consider a task set consisting of two task graphs T_1 and T_2. T_1 is a simple chain of three vertices with the first two vertices having their execution requirements equal to 1 and deadlines equal to 2, and the third vertex having an execution requirement of 3 and deadline equal to 6. The intertriggering separation on any directed edge (u, v) is equal to the deadline of u. T_2 consists of a single vertex having an execution requirement of 1 and deadline equal to 4. It can be seen that T_2 is indeed lowest-priority schedulable and passes the test of Theorem 8, but fails the test given by Theorem 6. Lastly, we show that for any set of exactly two task graphs, the test given by Theorem 8 is both a necessary and sufficient condition.

Thm. 9: *For any task set $\mathcal{T}$ consisting of exactly two task graphs, a task $T \in \mathcal{T}$ is lowest-priority schedulable if and only if it satisfies the test given by Theorem 8.*

Proof: Let $\mathcal{T}$ consist of two task graphs T_1 and T_2. By Lemma 1, we assume that T_1 consists of a single vertex with an execution requirement of e and deadline equal to d. T_2 is an arbitrary task graph. We want to test whether T_1 is lowest-priority schedulable. The claim is that T_1 is lowest-priority schedulable if and only if $\exists t \leq d$ for which $(t - T_2.rbf'(t)) \geq e$.

To prove that the condition is necessary, it is sufficient to prove that if $\forall t \leq d$, $(t - T_2.rbf'(t)) < e$ then T_1 is not lowest-priority feasible. The proof of this is straightforward and hence we omit the details. □

3.5 Algorithms for a restricted task model

In this section we consider a restricted form of the task model described in Section 3.2—we do not consider the recurring behaviour of the task graphs. The task graphs considered here can therefore be referred to as “one-shot” task graphs, where the control flows from the source to the sink vertex of a graph only once. For a collection of such *one-shot* task graphs, we show that the schedulability analysis problem can be solved in pseudo-polynomial time, and for the special case where all the vertices of a task graph have equal execution times, this problem can be solved in polynomial time. These results are then used to derive polynomial time approximate decision algorithms in Section 3.7.

3.5.1 Pseudo-polynomial time dynamic-priority schedulability analysis

Given a task graph T, we first give a pseudo-polynomial time algorithm for computing $T.dbf(t)$ for any $t \geq 0$, based on dynamic programming. Let there be n vertices in T denoted by $v_1, \ldots, v_n$, and without any loss of generality we assume that there can be a directed edge from v_i to v_j only if $i < j$. Following our notation described in Section 3.2, associated with each vertex v_i is its execution requirement $e(v_i)$ which here is assumed to be integral (a pseudo-polynomial algorithm is meaningful only under this assumption), and its deadline $d(v_i)$. Associated with each edge (v_i, v_j) is the minimum intertriggering separation $p(v_i, v_j)$.

Let $t_{i,e}$ be the minimum time interval within which the task T can have an execution requirement of exactly e time units due to some legal triggering sequence, considering only a subset of vertices from the set $\{v_1, \ldots, v_i\}$, if all the triggered vertices are to meet their respective deadlines. Let $t^i_{i,e}$ be the minimum time interval within which a sequence of vertices from the set $\{v_1, \ldots, v_i\}$, and ending with the vertex v_i, can have an execution requirement of exactly e time units, if all the vertices have to meet their respective deadlines. Lastly, let $E = \max_{i=1,\ldots,n} e(v_i)$. Clearly, nE is an upper bound on $T.dbf(t)$ for any $t \geq 0$. It can be shown by induction that Algorithm 1 correctly computes $T.dbf(t)$, and has a running time of $O(n^3E)$, and is hence a pseudo-polynomial time algorithm for computing the demand-bound function.

Theorem 3 along with Algorithm 1 implies a pseudo-polynomial time algorithm for dynamic-priority schedulability analysis. To see this, let for any task $T \in \mathcal{T}$, t^T_{max} denote the maximum amount of time elapsed among all execution sequences starting from the source vertex of T and ending at the sink vertex, if every vertex is triggered at the earliest possible time (respecting the minimum intertriggering separations). Let $t_{max} = \max_{T \in \mathcal{T}} t^T_{max}$. It follows from Theorem 3 that $\mathcal{T}$ is dynamic-priority schedulable if and only if $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ for all $t = 1, \ldots, t_{max}$. $T.dbf(t)$ for any t can be determined in pseudo-polynomial time by Algorithm 1 and clearly, t_{max} is pseudo-polynomially bounded, implying a pseudo-polynomial time algorithm for dynamic-priority schedulability analysis for our restricted (one-shot) task model.

Algorithm 1 Computing $T.dbf(t)$ in pseudo-polynomial time

```
Require: Task graph $T$, and a real number $t \geq 0$
  for $e \leftarrow 1$ to $nE$ do
    $t_{1,e} \leftarrow \begin{cases} d(v_1) & \text{if } e(v_1) = e \\ \infty & \text{otherwise} \end{cases}$
    $t^1_{1,e} \leftarrow t_{1,e}$
  end for
  for $i \leftarrow 1$ to $n-1$ do
    for $e \leftarrow 1$ to $nE$ do
      Let there be directed edges from the vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ to $v_{i+1}$
      $t^{i+1}_{i+1,e} \leftarrow \begin{cases} \min\{t^{i_j}_{i_j,e-e(v_{i+1})} - d(v_{i_j}) + p(v_{i_j}, v_{i+1}) \\ \qquad +d(v_{i+1}) \mid j = 1, \ldots, k\} \text{ if } e(v_{i+1}) < e, \\ d(v_{i+1}) \text{ if } e(v_{i+1}) = e, \text{ and } \infty \text{ otherwise} \end{cases}$
      $t_{i+1,e} \leftarrow \min\{t_{i,e}, t^{i+1}_{i+1,e}\}$
    end for
  end for
  $T.dbf(t) \leftarrow \max\{e \mid t_{n,e} \leq t\}$
```

Vertices with equal execution requirements: We now show that for the special case where for every (one-shot) task graph T belonging to a task set $\mathcal{T}$, all the vertices of T have equal execution requirements, the schedulability analysis problem for $\mathcal{T}$ can be solved in polynomial time, in contrast to the pseudo-polynomial time algorithm given above. This result holds even when all execution requirements and deadlines take values over the reals.

We denote the vertices of a task graph T by $v_1, \ldots, v_n$ and assume that there can be a directed edge from v_i to v_j only if $i < j$. Let $t_{i,k}$ denote the minimum time interval within which exactly k vertices of T from the set $\{v_1, \ldots, v_i\}$ (obviously $k \leq i$) need to be executed as a result of some legal triggering sequence, if they have to meet their associated deadlines. Let $t^i_{i,k}$ denote the minimum time interval within which exactly k vertices of T consisting of v_i and any other $k-1$ vertices from $\{v_1, \ldots, v_{i-1}\}$ need to be executed as a result of some legal triggering sequence, if they have to meet their associated deadlines.

Given any vertex v_i of T, let there be directed edges from the vertices $v_{i_1}, \ldots, v_{i_l}$ to v_i. Then for any $k \leq i$,

$$\begin{aligned} t^i_{i,k} &= \min\{t^{i_j}_{i_j,k-1} - d(v_{i_j}) + p(v_{i_j}, v_i) + d(v_i) \mid j = 1, \ldots, l\} \\ &\quad \text{(and equal to } d(v_i), \text{ if } k = 1) \\ t_{i,k} &= \min\{t_{i-1,k}, t^i_{i,k}\} \end{aligned}$$

Using the fact that $t_{1,1} = t^1_{1,1} = d(v_1)$, it is now possible to compute any $t_{i,k}$ within at most $O(n^3)$ time, where n is the number of vertices in the task graph. Now, if each task graph $T \in \mathcal{T}$ has n_T vertices then let us consider the set $S = \bigcup_{T \in \mathcal{T}} \bigcup_{i=1,\ldots,n_T} \{t_{n_T,i} \text{ for task graph } T\}$. If each vertex of task graph T has an execution requirement of e, then for ant $t \geq 0$, $T.dbf(t) = \max\{ie \mid t_{n_T,i} \leq t\}$. Clearly, the task set $\mathcal{T}$ is schedulable if and only if $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ for all $t \in S$. Computing all the necessary $T.dbf(t)$ values for each task graph T and storing them in a table takes $O(n^3)$ time if the number of vertices in T is bounded by $O(n)$. Since there are $|\mathcal{T}|$ task graphs, this whole process takes $O(|\mathcal{T}|n^3)$ time. For each value of t, verifying whether the sum of the demand-bound functions exceeds t requires a search through the previously computed tables and takes $O(|\mathcal{T}| \log n)$ time. Since there are $O(|\mathcal{T}|n)$ values of t for which this has to be verified, this takes $O(|\mathcal{T}|^2 n \log n)$ time. Hence the total run time is bounded by $O(|\mathcal{T}|n^3 + |\mathcal{T}|^2 n \log n)$ which is polynomial in the size of the input specification.

3.5.2 Pseudo-polynomial time static-priority schedulability analysis

Based on our new definition of the resource-bound function $T.rbf'(t)$, which we introduced in Section 3.4.4, in this section we give a pseudo-polynomial time algorithm for static-priority schedulability analysis for our restricted one-shot task model. This algorithm is based on a pseudo-polynomial time algorithm for computing $T.rbf'(t)$, similar to Algorithm 1 given in Section 3.5.1.

Following the notation used in Section 3.5.1, given a task graph T, let $t_{i,e}$ denote the minimum time interval within which T can have an execution requirement of exactly e time units due to some legal triggering sequence, considering only a subset of vertices from the set $\{v_1, \ldots, v_i\}$. Let $t^i_{i,e}$ be the minimum time interval within which any execution sequence consisting of vertices from the set $\{v_1, \ldots, v_{i-1}\}$ and ending with the vertex v_i can have an execution requirement of exactly e time units. Now recall the definition of $t^i_{i,e}$ as used in Section 3.5.1 for computing $T.dbf(t)$, which is the minimum time interval within which a sequence of vertices from the set $\{v_1, \ldots, v_i\}$, and ending with the vertex v_i can have an execution requirement of exactly e time units, if all the vertices have to meet their respective deadlines. This we denote here by $dbf^i_i(e)$. We assume, as in Section 3.5.1, that T consists of n vertices $v_1, \ldots, v_n$ and that there can be a directed edge from v_i to v_j only if $i < j$, and that all the execution requirements are integral. If $E = \max_{i=1,\ldots,n} e(v_i)$, then Algorithm 2 correctly computes $T.rbf'(t)$ and has a running time of $O(n^3E^2)$.

It now follows from Theorem 5, Algorithm 2 and Theorem 8, that there exists a pseudo-polynomial algorithm for static-priority schedulability analysis for a sollection of one-shot task graphs, that implements the sufficiency condition stated by Theorem 8. Lastly, in the case where all the vertices of a task graph have equal execution times, this problem can also be solved in polynomial time following exactly the same approach as used in Section 3.5.1 for dynamic-priority schedulability analysis.

Algorithm 2 Computing $T.rbf'(t)$ in pseudo-polynomial time

```
Require: Task graph $T$, and a real number $t \geq 0$
  for $e \leftarrow 1$ to $nE$ do
    $t_{1,e} \leftarrow \begin{cases} e & \text{if } e \leq e(v_1) \\ \infty & \text{if } e > e(v_1) \end{cases}$
    $t^1_{1,e} \leftarrow t_{1,e}$
  end for
  Computing $t_{i+1,e}$:
    Let there be directed edges from the vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ to $v_{i+1}$
    Let $t^{i_j,i+1}_{i+1,e}(l) \leftarrow dbf^{i_j}_{i_j}(e - e(v_{i+1}) + l) - d(v_{i_j}) + p(v_{i_j}, v_{i+1}) + e(v_{i+1}) - l$
    Let $t^{i_j,i+1}_{i+1,e} \leftarrow \min\{t^{i_j,i+1}_{i+1,e}(l) \mid l = 0, \ldots, e(v_{i+1}) - 1\}$
    $t^{i+1}_{i+1,e} \leftarrow \min\{t^{i_j,i+1}_{i+1,e} \mid j = 1, \ldots, k\}$
    $t_{i+1,e} \leftarrow \min\{t_{i,e}, t^{i+1}_{i+1,e}\}$
  $T.rbf'(t) \leftarrow \max\{e \,|\, t_{n,e} \leq t\}$
```

3.6 Schedulability with bounds on preemptions

The conditions presented in Sections 3.4.1 and 3.4.2 test the schedulability of a task set in a preemptive setup, i.e. the execution of a vertex in a task graph might be preempted at any time, in favour of a vertex belonging to a different task graph. Although non-preemptive scheduling is more restrictive, in many embedded system scenarios the advantage obtained using by using a preemptive scheduler is offset by the overheads involved in terms of more memory requirements and greater implementation complexity. This is especially true in the context of the problem studied in this thesis, because preempting the processing of a packet is generally too costly both in terms of memory and the context switching time and hence the advantage of preemption can not be exploited. Most network packet processors rather use hardware-supported multithreading.

In this section we present the conditions for schedulability under a non-preemptive setup, in the sense described in Section 3.2, i.e. once the vertex of a task graph starts executing, it can not be preempted, but when it completes execution then a vertex from a different graph can be scheduled for execution. As in Section 3.5, here too we consider the restricted version of the task model where the control flow in any task graph is only from the source to the sink vertex and the recurring behaviour of a task graph is not considered. The conditions derived here for this restricted model can be extended using the same techniques described in Section 3.4.3.

We first give a necessary and sufficient condition for the schedulability of a set of task graphs under EDF scheduling which, as described in Section 3.2.3, is an optimal dynamic-priority non-preemptive scheduling policy if the scheduler is work conserving. Not surprisingly, we show that for our task model, EDF is also an optimal non-preemptive scheduler. The condition derived here is substantially more complex compared to the one for schedulability under unbounded number of preemptions (given by Theorem 3), and is specified by Algorithm 3. This algorithm uses the demand-bound function $T.dbf(t)$, and a similar function $T.dbf^v(t)$ which was introduced in Section 3.4.1. The correctness of the algorithm is given by Theorem 10.

Algorithm 3 Algorithm for schedulability analysis under non-preemptive EDF

```
Require: Task set $\mathcal{T}$
$decision \leftarrow$ SCHEDULABLE
for all tasks $T_i \in \mathcal{T}$ and for all vertices $v \in T_i$ and for all $\hat{\tau} \geq 0$ do
    Let $\tilde{\mathcal{T}} \leftarrow \mathcal{T} \backslash \{T_i\}$
    $\mathcal{T}_{dbf=0} \leftarrow \{T \in \tilde{\mathcal{T}} \mid T.dbf(\hat{\tau} + d(v)) = 0\}$
    $e_{max} \leftarrow \max_{v'}\{e(v') \mid v'$ is a vertex of a task $T \in \mathcal{T}_{dbf=0}\}$
    Let $\mathcal{T}_{dbf>0} \leftarrow \{T \in \tilde{\mathcal{T}} \mid T.dbf(\hat{\tau} + d(v)) > 0\}$ and $q \leftarrow |\mathcal{T}_{dbf>0}|$
    $index \leftarrow 0$
    for $p \leftarrow 1$ to $q$ do
        Let $e'_{max} \leftarrow \max\{e(v') \mid v' \in T_p,\ d(v') > \hat{\tau} + d(v)\}$
        if $index = 0$ then
            if $e'_{max} > (T_p.dbf(\hat{\tau} + d(v)) + e_{max})$ then
                $e_{max} \leftarrow e'_{max}$
                $index \leftarrow p$
            end if
        else
            if $e'_{max} + T_{index}(\hat{\tau} + d(v)) > (T_p.dbf(\hat{\tau} + d(v)) + e_{max})$ then
                $e_{max} \leftarrow e'_{max}$
                $index \leftarrow p$
            end if
        end if
    end for
    if $index \neq 0$ then
        $\hat{\mathcal{T}} \leftarrow \mathcal{T}_{dbf>0} \backslash \{T_{index}\}$
    end if
    if $\hat{\tau} + d(v) < (T_i.dbf^v(\hat{\tau} + d(v)) + \sum_{T \in \hat{\mathcal{T}}} T.dbf(\hat{\tau} + d(v)) + e_{max})$ then {Condition (†)}
        $decision \leftarrow$ NOT SCHEDULABLE
    end if
end for
return $decision$
```

Thm. 10: *A task set $\mathcal{T}$ is non-preemptively schedulable under EDF if and only if Algorithm 3 returns* SCHEDULABLE.

Proof: Let v be any vertex of a task graph $T_i \in \mathcal{T}$. The vertex v has an execution requirement of $e(v)$ and a deadline equal to $d(v)$. Let v be triggered at time t and it completes execution at time $t + \delta$.

Let $R_{\mathcal{T}}^{\leq d}[t, t+\tau]$ denote the sum of the execution requirements of the vertices of any task graph $T \in \mathcal{T}$ which have been triggered in the time interval $[t, t+\tau]$

and which have their deadlines less than or equal to d. Let $W^{v,t}(t+\tau)$ $(0 \leq \tau \leq \delta)$ denote the total execution requirement at time $t+\tau$ that was generated by all the tasks in $\mathcal{T}$, and which must be met by the processor (under EDF scheduling) before the vertex v that was triggered at time t can complete its execution. $W^{v,t}(t+\tau)$ includes the execution requirement $e(v)$ of the vertex v as well. We assume that the processor was idle before time 0.

If we look back in time, let $t-\hat{\tau}$ be the first time before the time instant t when the processor does not have any vertex to execute with deadline less than or equal to $t+d(v)$ (i.e. the deadline of the vertex v). Hence, during the entire interval $[t-\hat{\tau}, t+\delta)$, the processor always has some vertex to execute with deadline less than or equal to $t+d(v)$. $W^{v,t}(t+\tau)$ for any $0 \leq \tau \leq \delta$ is therefore composed of the following: (1) The remaining execution requirement of the vertex that is in execution at time $t-\hat{\tau}$, denoted by $P(t-\hat{\tau})$. By our assumption of $\hat{\tau}$, the deadline of this vertex is greater than $t+d(v)$. (2) The execution requirement generated by the vertices of the task T_i during the time interval $[t-\hat{\tau}, t]$. This includes the vertex v. Clearly, all these vertices have a deadline less than or equal to $t+d(v)$. Therefore, this equals to $R_{T_i}^{\leq t+d(v)}[t-\hat{\tau}, t]$. (3) The execution requirement generated by vertices with deadlines less than or equal to $t+d(v)$, from all tasks belonging to a set, say $\hat{\mathcal{T}}$, where $\hat{\mathcal{T}} \subseteq \mathcal{T}\backslash\{T_i\}$, during the time interval $[t-\hat{\tau}, t+\tau]$. Therefore, this is equal to $\sum_{T\in\hat{\mathcal{T}}} R_T^{\leq t+d(v)}[t-\hat{\tau}, t+\tau]$. (4) The execution requirement served by the processor during the time interval $[t-\hat{\tau}, t+\tau]$.

Since we are considering a non-preemptive environment, the vertex which is in execution at the time $t-\hat{\tau}$ has to finish executing before any vertex having a deadline less or equal to $t+d(v)$ can be executed. Therefore, the processor always executes some vertex having a deadline less than or equal to $t+d(v)$ during the interval $[t-\hat{\tau}+P(t-\hat{\tau}), t+\tau]$. Hence,

$$\begin{aligned} W^{v,t}(t+\tau) &= P(t-\hat{\tau}) + R_{T_i}^{\leq t+d(v)}[t-\hat{\tau}, t] + \\ &\quad \sum_{T\in\hat{\mathcal{T}}} R_T^{\leq t+d(v)}[t-\hat{\tau}, t+\tau] - (\hat{\tau}+\tau) \end{aligned} \tag{3.3}$$

Now, note that if there exists a τ $(0 \leq \tau \leq d(v))$ such that $W^{v,t}(t+\tau) = 0$, then the vertex v completes execution on or before its deadline. Substituting $\tau = d(v)$ in Equation (3.3), we obtain:

$$\begin{aligned} W^{v,t}(t+d(v)) &= P(t-\hat{\tau}) + R_{T_i}^{\leq t+d(v)}[t-\hat{\tau}, t] + \\ &\quad \sum_{T\in\hat{\mathcal{T}}} R_T^{\leq t+d(v)}[t-\hat{\tau}, t+d(v)] - \hat{\tau} - d(v) \end{aligned}$$

Following our definition of the *demand-bound functions* (dbf and dbf^v), clearly,

$$\begin{aligned} W^{v,t}(t+d(v)) &\leq P(t-\hat{\tau}) + T_i.dbf^v(\hat{\tau}+d(v)) + \\ &\quad \sum_{T\in\hat{\mathcal{T}}} T.dbf(\hat{\tau}+d(v)) - \hat{\tau} - d(v) \end{aligned} \tag{3.4}$$

To compute an upper bound on $W^{v,t}(t + d(v))$ we would like to maximize the right hand side of the above inequality (3.4). For this, note that if a vertex v' of a task T contributes to the term $P(t - \hat{\tau})$, then T can not belong to the set $\hat{\mathcal{T}}$. Following this constraint, for any task T_i and any vertex $v \in T_i$, Algorithm 3 computes $P(t - \hat{\tau}) = e_{max}$ and the task set $\hat{\mathcal{T}}$ which maximizes the right hand side of Inequality (3.4). Therefore, if the algorithm returns SCHEDULABLE, then we have (from Condition (†) of the algorithm),

$$W^{v,t}(t + d(v)) \leq \hat{\tau} + d(v) - (\hat{\tau} + d(v)) = 0$$

Hence, there exists a $\tau \leq t + d(v)$ such that $W^{v,t}(t + d(v)) \leq 0$ and therefore the vertex v completes execution before its deadline.

Now we give the proof of necessity. Suppose that for some task $T_i \in \hat{\mathcal{T}}$ and for some vertex $v \in T_i$ and for some $\hat{\tau}$, the Condition (†) in Algorithm 3 holds. We claim that in this case the task set $\mathcal{T}$ is not feasible. The term e_{max} in Condition (†) is due to some vertex v' of some task in $\mathcal{T}$ (not equal to T_i) and $d(v') > d(v) + \hat{\tau}$. Assume that the processor is empty before time $t - \hat{\tau}$ and just before $t - \hat{\tau}$ the vertex v' is triggered. Starting from time $t - \hat{\tau}$ all the tasks $T \in \hat{\mathcal{T}}$ generate an execution requirement due to a sequence of vertex triggerings which are the same as those which result in the computation of $T.dbf(\hat{\tau} + d(v))$ in Condition (†) of Algorithm 3. The task T_i also generates an execution requirement due to a sequence of triggerings that result in $T_i.dbf^v(\hat{\tau} + d(v))$ in Condition (†), starting from the time $t - \hat{\tau}$, with the vertex v being triggered at time t.

Therefore, the execution requirement that has still to be met by the processor at time $t + d(v)$, before the vertex v can complete execution, is given by:

$$\begin{aligned} W^{v,t}(t + d(v)) &= e_{max} + T_i.dbf^v(\hat{\tau} + d(v)) + \\ &\quad \sum_{T \in \hat{\mathcal{T}}} T.dbf(\hat{\tau} + d(v)) - (\hat{\tau} + d(v)) \end{aligned}$$

Hence, from Condition (†) in Algorithm 3, we obtain that $W^{v,t}(t + d(v)) > \hat{\tau} + d(v) - (\hat{\tau} + d(v)) = 0$. Since apart from vertex v' (which can not be preempted), all the vertices of T_i and all the vertices of the tasks in $\hat{\mathcal{T}}$ that have been triggered have a deadline of less than or equal to $t + d(v)$, some vertex misses its deadline at $t + d(v)$. □

The optimality of EDF as a dynamic-priority non-preemptive work conserving scheduler follows from the fact that the proof of necessity makes no assumptions about the scheduling discipline.

We next show that using the pseudo-polynomial time algorithm for computing the demand-bound function (i.e. Algorithm 1), Algorithm 3 also runs in pseudo-polynomial time. Note that Algorithm 1 can also be used to compute the function $T.dbf^v(t)$. Following the same notation as used in this algorithm, for any vertex v_i of a task graph T, $T.dbf^{v_i}(t) = \max\{e \mid t^i_{i,e} \leq t\}$.

As in Section 3.5.1, let for any task $T \in \mathcal{T}$, t^T_{max} denote the maximum amount of time elapsed among all execution sequences starting from the source vertex of T and ending at the sink vertex, if every vertex is triggered at the earliest possible time (respecting the minimum intertriggering separations). Let $t_{max} = \max_{T \in \mathcal{T}} t^T_{max}$. Clearly, it is sufficient to test the Condition (†) in Algorithm 3 only for $\hat{\tau} = 1, \ldots, t_{max}$. Both $T.dbf(\hat{\tau} + d(v))$ and $T.dbf^v(\hat{\tau} + d(v))$ in the Step 25 of the algorithm for any $\hat{\tau}$, (and the values of $T.dbf(t)$ in other steps of the algorithm) can be determined in pseudo-polynomial time by Algorithm 1 and clearly, t_{max} is pseudo-polynomially bounded, implying a pseudo-polynomial algorithm for schedulability analysis.

We next present a condition for schedulability, for non-preemptive static-priority schedulers. As described in Section 3.4.2, here, a priority is assigned to each task graph, and among the ready vertices the scheduler always selects a vertex belonging to the highest priority task. Unlike the case with EDF, the schedulability test that we derive here is only a sufficient but not a necessary condition (as was the case in the preemptive setup described in Sections 3.4.2 and 3.4.4). Further, it only gives a means for testing whether a given static-priority assignment to the different tasks results in the task set being schedulable. This is unlike the conditions given in Sections 3.4.2 and 3.4.4, which gave a means for testing "whether there exists *any* static-priority assignment to the tasks under which the task set is static-priority schedulable". Here the test is based on the request-bound function $T.rbf(t)$ as defined in Section 3.4.2, and is given by Theorem 11. Using the same technique as used in computing the modified request-bound function $T.rbf'(t)$ in Algorithm 2, $T.rbf(t)$ for any t can also be computed in pseudo-polynomial time, thereby giving a pseudo-polynomial time algorithm for testing the condition given by Theorem 11.

Thm. 11: *Given a task set $\mathcal{T} = \{T_1, \ldots, T_k\}$, where the task T_p has priority p $(1 \leq p \leq k)$ and $p < q$ indicates that T_p has a higher priority than T_q. The task set $\mathcal{T}$ is static-priority schedulable if for all tasks T_p the following condition holds: for all vertices v belonging to the task graph of T_p, and for all $t \geq 0$, $\exists\, 0 \leq \tau \leq d(v) - e(v)$ for which*

$$t + \tau \geq T_p.rbf(t) + \sum_{q=1}^{p-1} T_q.rbf(t + \tau) - e(v) + e^{max}_{>p}$$

where $e^{max}_{>p} = \max\{e(v') \mid v'$ is a vertex in any of the task graphs $T_l,\ l = p+1, \ldots, q\}$

Proof: Let v be any vertex of the priority-p task graph T_p. The vertex v has an execution requirement of $e(v)$ and a deadline equal to $d(v)$. Let v be triggered at time t and it completes execution at time $t + \delta$.

Let $W^{v,t}(t + \tau)$ $(0 \leq \tau \leq \delta)$ denote the total execution requirement at time $t + \tau$ that was generated by all the task in $\mathcal{T}$, and which must be met before the vertex v that was triggered at time t can complete its execution. $W^{v,t}(t + \tau)$ therefore includes the execution requirement $e(v)$ of the vertex v as well.

Now if we look back in time, let $t - \hat{\tau}$ be the first time before the time instant t when the processor did not have any vertex of any task graph of priority

$\leq p$ to execute. Clearly, $t - \hat{\tau}$ is the time instant at which some vertex of a task graph having priority $\leq p$ was triggered. The processor at this time was either executing some vertex of a task graph having priority $> p$ or was idle. $W^{v,t}(t + \tau)$ is therefore composed of the following: (1) The remaining execution requirement of some vertex of a task graph having priority $> p$, (2) The execution requirement generated by vertices of the task graph T_p (including the vertex v) during the time interval $[t - \hat{\tau}, t]$, (3) The execution requirement generated by vertices of task graphs $\cup_{q=1}^{p-1}\{T_q\}$ during the time interval $[t - \hat{\tau}, t + \tau]$, (4) The execution requirement served by the processor during the time interval $[t - \hat{\tau}, t + \tau]$.

Therefore,

$$W^{v,t}(t + \tau) \leq e^{max}_{>p} + T_p.rbf(\hat{\tau}) + \sum_{q=1}^{p-1} T_q.rbf(\tau + \hat{\tau}) - (\hat{\tau} + \tau) \qquad (3.5)$$

From the condition given in the theorem, we obtain that for $\hat{\tau}$, $\exists\, 0 \leq \tau' \leq d(v) - e(v)$ for which

$$\hat{\tau} + \tau' \geq T_p.rbf(\hat{\tau}) + \sum_{q=1}^{p-1} T_q.rbf(\hat{\tau} + \tau') - e(v) + e^{max}_{>p}$$

Using this in the Expression (3.5) implies that $\exists\, 0 \leq \tau' \leq d(v) - e(v)$ for which

$$W^{v,t}(t + \tau') \leq (\hat{\tau} + \tau') + e(v) - (\hat{\tau} + \tau') = e(v)$$

Now since $W^{v,t}(t + \tau')$ includes $e(v)$, the execution requirement of the vertex v, either v is in execution at time $t + \tau'$ or it has already completed execution by this time. Hence v meets its deadline. □

3.7 Approximate schedulability analysis

All the algorithms for schedulability analysis presented till now are either of exponential complexity, or at best run in pseudo-polynomial time (except in very special cases with all the vertices in a task graph having the same execution requirement, where the algorithm runs in polynomial time). In view of the hardness results presented in Theorem 1 and Corollary 1, pseudo-polynomial time algorithms are the best that can be obtained in terms of *exact decision algorithms*. However, the run times of these algorithms are still prohibitive for even reasonably sized problems (as we show in Section 3.8), making them infeasible for use within any design or verification tool.

The work presented in this section remedies this situation to a large extent. It is based on the observation that if a small amount of error in the decisions made by a schedulability analysis algorithm is acceptable, then it is possible to design *approximate decision algorithms*, which run in polynomial time. This idea is similar in spirit to obtaining approximation algorithms for NP-hard optimization problems [79]. Algorithms for *approximate schedulability analysis* are of the following form: If a task set is schedulable then the algorithm is guaranteed to return the correct answer SCHEDULABLE. But if a task set is not schedulable then in some cases the algorithm might incorrectly return SCHEDULABLE as well. However, in such cases, it is guaranteed that no job can miss its deadline by a time interval which is "too large". The maximum length of time by which a job can miss its deadline (in case the algorithm incorrectly returns SCHEDULABLE) is bounded and is parameterized by an input error parameter ε. The smaller the value of ε, the smaller is this time interval and the higher is the running time of the algorithm. Therefore, ε represents a tradeoff between the maximum error that can be incurred and the running time of the algorithm.

In contrast to such *optimistic algorithms*, it is also possible to design *pessimistic algorithms* which always return the correct answer if a task set is not schedulable. However, if a task set is schedulable then the algorithm might err and incorrectly return NOT SCHEDULABLE. As in the previous case, the error incurred by such wrong decisions is bounded and is parameterized by ε. This means that task sets for which the algorithm incorrectly returns NOT SCHEDULABLE can load the processor "heavily", in the sense that there exists time intervals over which the processor might be almost always occupied if the deadline of all jobs are to be met. Here, the length of time, within such time intervals, for which the processor can be idle is a measure of the error incurred. As before, the smaller the value of ε, the smaller is the error, but at the expense of increasing the running time.

We also show that it is possible to give a third class of algorithms which can incur a double-sided error, meaning that both SCHEDULABLE and NOT SCHEDULABLE answers can be wrong. However, for such algorithms the maximum error in either direction is less than the error incurred for the equivalent optimistic and pessimistic algorithms.

In addition to polynomial time algorithms for approximate schedulability analysis for the task model described in Section 3.2 (and also for the recurring real-time task model [15]), it is also possible to design such algorithms for various other well known task models such as the sporadic model due to Mok [105, 17], the multiframe model of Mok and Chen [107] and the generalized multiframe model of Baruah *et al.* [16]. The known algorithms for (exact) schedulability analysis for all of these models have either pseudo-polynomial or exponential running time. Besides this, the other rationale behind approximate schedulability analysis is that in many embedded system scenarios, including the case of network packet processors, it is difficult to evaluate the worst-case execution times of tasks accurately. This is due to factors such as caching and pipelining in embedded processors. In such cases, either the worst-case execution times of tasks are overestimated, or it is acceptable for jobs generated by these tasks to miss their deadlines by small amounts of time. In either case, an approximate schedulability analysis, in the sense we described above, would suffice for all practical purposes. In other domains such as multimedia applications, although the execution requirements of tasks may be accurately determined, if a job misses its deadline by a small amount of time then the performance of the system (quality of audio or video) does not deteriorate too much.

This section is organized as follows. In the next subsection, we outline an abstract model for task systems. All taks models which fit into this abstract model are amenable to approximate schedulability analysis. We then describe the algorithms for approximate schedulability analysis for the task model studied in this chapter (as described in Section 3.2), following which we briefly outline how these algorithms extend to other task models such as the sporadic, multiframe, and generalized multiframe. In this section, we only address the dynamic-priority schedulability analysis problem. The results presented here also extend to the static-priority case, but to avoid being repetative we do not detail them here.

3.7.1 An abstract model of task systems

Here we present an abstract model of task systems based on certain *task-independence assumptions* given in [16]. These assumptions are extremely general and are satisfied by many task systems encountered in practice, including the one studied in this chapter. We then show that it is possible to derive polynomial time algorithms for approximate schedulability analysis for all such task systems. To show this, first we outline a generic framework for (exact) schedulability analysis for task models satisfying the conditions imposed by the abstract model, and based on this framework we then outline two basic building blocks which make up our algorithms for approximate schedulability analysis. Concrete examples of these two building blocks given in the context of the our task model in Section 3.7.2.

A *task* in this abstract model generates a (possibly infinite) sequence of *jobs*. Each job is characterized by a *ready-time*, an *execution requirement*, and a *deadline* (in the same sense as descrined in the begining of Section 3.2). A task set consists of a collection of such tasks, all of which are to be executed on a single shared processor and jobs are preemptable. The extension to a multiprocessor system can be done following the same techniques described in Section 3.2.1. The generation of jobs by a task is constrained by a set of rules, which for example might be that there is a minimum separation in time between the generation of two consecutive jobs by a task. Jobs generated according to these constraints are said to be *legal*. The schedulability analysis of a given task set, as described in Section 3.2.2, is concerned with determining whether it is possible to assign to each job a processor time equal to its execution requirement between its ready time and its deadline, for all possible legal job sequences generated by tasks of the task set.

The rules that govern the generation of jobs by a task can be stated in the form of the following two *task independence assumptions*. (i) The runtime behavior of a task is independent of any other tasks in the system. (ii) The constraints according to which legal job sequences are generated can be specified without any references to *absolute time*. Assumption (i) states that each task generates jobs independently of the jobs generated by other tasks in the system. Therefore, it is not permissible, for example, to require a task to generate a job in response to a job generated by another task. Assumption (ii) states that all temporal specifications defining the rules according to which jobs are generated by a task can only be relative to the time at which the task begins execution, or can be relative to the ready-time another job of the same task. Therefore, a constraint like the ready-times of two consecutive jobs of a task must be separated by at least p time units, conforms to this requirement. Lastly, the time at which a task begins execution (i.e. the first job is generated) is not *a priori* known. For example, a task can begin execution in response to some external event.

Note that although the task independence assumptions restrict the job generation process of a task (for example, by specifying the minimum separation between the generation of two jobs), they make no assumptions about the interactions between the jobs once they are generated. Once a job is generated, it executes independently of any other job in the system, including those generated by the same task.

Given a sequence of jobs generated by a task set $\mathcal{T}$, $[(T_i, a_i, e_i, d_i), (T_j, a_j, e_j, d_j), \ldots]$ ($T_i \in \mathcal{T}$ refers to a task, a_i is the ready time of a job, e_i is its execution requirement, and d_i is its deadline), the task independence assumptions imply that the sequence is legal if and only if all subsequences formed by jobs from the individual tasks are also legal (follows from Assumption (i)). Assumption (ii) implies that if $[(a_1, e_1, d_1), (a_2, e_2, d_2), \ldots]$ is a legal sequence of jobs generated by a task, then the sequence $[(a_1 - t, e_1, d_1 - t), (a_2 - t), e_2, d_2 - t), \ldots]$ is also legal, where t is any real number.

It may be verified that these assumptions are followed our task model described in Section 3.2 and also by a wide variety of other task models such as the sporadic model, the multiframe model, the generalized multiframe model, and the recurring real-time task model.

Exact schedulability analysis: For this abstract task model, it may be seen that the condition for schedulability under preemption as given by Theorem 3 holds (because, the proof of this theorem does not make use of any feature which is specific to the task model described in Section 3.2 , and which does not hold for the abstract task model).

Therefore, a task set $\mathcal{T}$ is not schedulable if and only if there exists some $\hat{t}$ for which $\sum_{T \in \mathcal{T}} T.dbf(\hat{t}) > \hat{t}$. The schedulability analysis algorithms for task models such as sporadic, generalized multiframe, and the recurring real-time task model are based on identifying an upper bound $t_{\max}$, such that if $\mathcal{T}$ is not schedulable then there exists some $\hat{t} \leq t_{\max}$ for which $\sum_{T \in \mathcal{T}} T.dbf(\hat{t}) > \hat{t}$. Hence, a schedulability analysis algorithm is based on checking if $\sum_{T \in \mathcal{T}} T.dbf(t)$ is greater than t for all $t \leq t_{\max}$. If $\sum_{T \in \mathcal{T}} T.dbf(t)$ is less than or equal to t for all such values of t then the algorithm returns

SCHEDULABLE, else it returns NOT SCHEDULABLE. Recall from Corollary 2 that for our task model, $t_{\max}$ is equal to $\frac{\sum_{T \in \mathcal{T}} 2E(T)}{1-\sum_{T \in \mathcal{T}} \frac{E(T)}{P(T)}}$, where $E(T)$ and $P(T)$ are as described in Section 3.4.1.

However, for all of the above models, $t_{\max}$ turns out to be pseudo-polynomial in the size of the input specification. Additionally, in our task model, the problem of computing the value of $T.dbf(t)$ for any t is NP-hard and also requires pseudo-polynomial time. Hence, the overall algorithm for schedulability analysis for all these models run in pseudo-polynomial time.

3.7.1.1 A framework for approximate schedulability analysis

In its most general form, our framework for approximate schedulability analysis relies on the following two building blocks (see Figure 14) (i) Obtaining an approximation algorithm to compute the demand-bound function $T.dbf(t)$ for any task graph T and time interval of length t in polynomial time. (ii) Instead of checking the value of $\sum_{T \in \mathcal{T}} T.dbf(t)$ for all $t \leq t_{\max}$ (which can be pseudo-polynomial number of checks), only a polynomial number of checks are done.

Both the above two steps result in some *error*. The main contribution of the work in this section is to show that if an appropriate polynomial time approximation algorithm exists for the first step then the *total error* incurred by the approximate schedulability analysis from the two steps is bounded (in a sense that we describe later) and it is possible to obtain a tradeoff between this error and the running time of the algorithm.

Let us assume that an approximation algorithm for computing the demand-bound function $T.dbf(t)$ exists and it takes as an input an error parameter ε and in polynomial time returns for any t an approximate value of the function denoted by $T.dbf'(t)$, such that

$$T.dbf(t) \geq T.dbf'(t) \geq f(\varepsilon)T.dbf(t)$$

where f is some function of ε. Hence, we have

$$\frac{1}{f(\varepsilon)}T.dbf'(t) \geq T.dbf(t) \geq T.dbf'(t) \tag{3.6}$$

For example, if the approximation algorithm is a fully-polynomial time approximation scheme (FPTAS) [79] then

$$T.dbf \geq T.dbf'(t) \geq (1-\varepsilon)T.dbf(t)$$

and hence the above Inequality (3.6) takes the form

$$\frac{1}{1-\varepsilon}T.dbf'(t) \geq T.dbf(t) \geq T.dbf'(t)$$

If the size of the input specification of the task T is $O(n)$ and $0 < \varepsilon < 1$, then such an FPTAS for computing $T.dbf'(t)$ for any t runs in $poly(n, \frac{1}{\varepsilon})$ time (where $poly$ denotes some polynomial function) and the smaller the value of ε

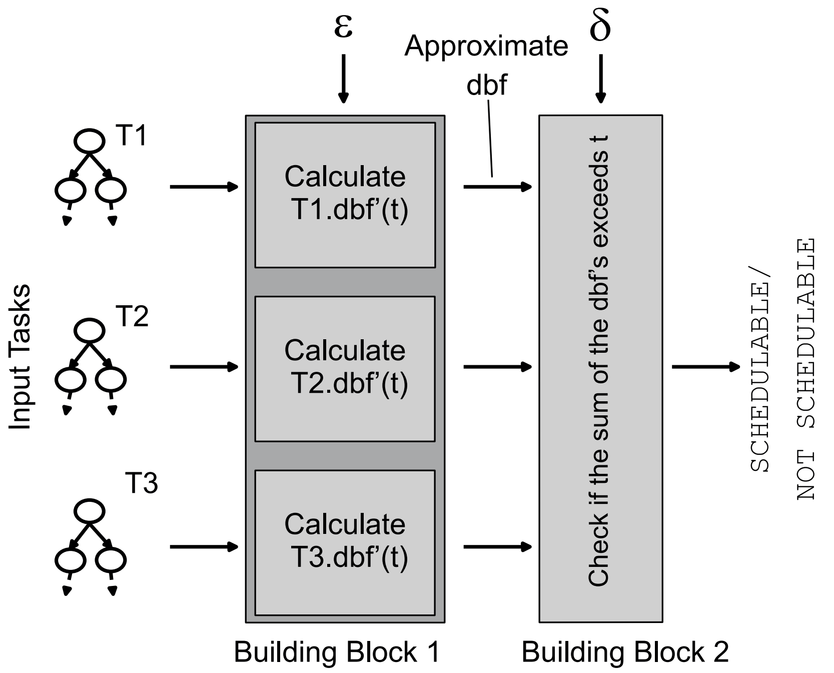

Fig. 14: A framework for approximate schedulability analysis. Building Block 1 approximately computes the demand-bound function. Building Block 2 performs a polynomial number of checks to verify if the sum of the demand-bound functions for any t exceeds t. ε and δ are error parameters given as input to the algorithm.

the less is the error in estimating $T.dbf(t)$, however, at the cost of increasing the running time.

Hence, for any $t \geq 0$, $\frac{1}{f(\varepsilon)}T.dbf'(t)$ and $T.dbf'(t)$ can be used as upper and lower bounds for $T.dbf(t)$ and such bounds can be computed in polynomial time. Our approximate schedulability analysis is based on using either this upper or lower bound for $T.dbf(t)$ to check if the sum of the demand-bound functions for all the tasks in the task set $\mathcal{T}$ exceeds t for any value of $t \leq t_{\max}$.

Let us first suppose that we use the lower bound $T.dbf'(t)$, and for any $t \leq t_{\max}$ return NOT SCHEDULABLE if $\sum_{T\in\mathcal{T}} T.dbf'(t) > t$. If $\sum_{T\in\mathcal{T}} T.dbf'(t) \leq t$, for all $t \leq t_{\max}$ then we return SCHEDULABLE. Such an algorithm is overly *optimistic* in the sense that if a task set $\mathcal{T}$ is schedulable then the algorithm is guaranteed to return the correct answer. However, for some task sets the algorithm might incorrectly return SCHEDULABLE even if they are not. In such cases, for some $t \leq t_{\max}$, $\sum_{T\in\mathcal{T}} T.dbf(t) > t$ but $\sum_{T\in\mathcal{T}} T.dbf'(t) \leq t$.

Therefore, the error incurred is equal to

$$\begin{aligned}
& \sum_{T\in\mathcal{T}} T.dbf(t) - \sum_{T\in\mathcal{T}} T.dbf'(t) \\
\leq\ & \sum_{T\in\mathcal{T}} T.dbf(t) - f(\varepsilon)\sum_{T\in\mathcal{T}} T.dbf(t) \\
=\ & (1 - f(\varepsilon))\sum_{T\in\mathcal{T}} T.dbf(t)
\end{aligned}$$

Hence, for such a value of t, a job can miss its deadline by at most $(1 - f(\varepsilon))\sum_{T\in\mathcal{T}} T.dbf(t)$ time units (in the case of an FPTAS for approximating $T.dbf(t)$, this is equal to $\varepsilon\sum_{T\in\mathcal{T}} T.dbf(t)$). For small values of ε, such an algorithm can therefore make an error only for task sets in which jobs can miss their deadlines only by small intervals of time.

Alternatively, it is possible to design a *pessimistic algorithm*, which for a task set $\mathcal{T}$ returns `SCHEDULABLE` if $\sum_{T\in\mathcal{T}} \frac{1}{f(\varepsilon)} T.dbf'(t) \leq t$, for all $t \leq t_{\max}$, else it returns `NOT SCHEDULABLE`. Since $\frac{1}{f(\varepsilon)} T.dbf'(t)$ is an overestimate of $T.dbf(t)$, such an algorithm always returns the correct answer if a task set is not schedulable. However, for certain task sets which are not schedulable, this algorithm might return an incorrect answer. In such cases, for some $t \leq t_{\max}$, $\sum_{T\in\mathcal{T}} T.dbf(t) \leq t$ but $\sum_{T\in\mathcal{T}} \frac{1}{f(\varepsilon)} T.dbf'(t) > t$. The error incurred is therefore equal to

$$\sum_{T\in\mathcal{T}} \frac{1}{f(\varepsilon)} T.dbf'(t) - \sum_{T\in\mathcal{T}} T.dbf(t) \quad\leq\quad \frac{1-f(\varepsilon)}{f(\varepsilon)} \sum_{T\in\mathcal{T}} T.dbf(t)$$

Therefore, task sets for which this algorithm can err are those which over some time interval of length t can load the processor for at least $t - \frac{1-f(\varepsilon)}{f(\varepsilon)} \sum_{T\in\mathcal{T}} T.dbf(t)$ time units (which in the case of the FPTAS for approximating $T.dbf(t)$ is $t - \frac{\varepsilon}{1-\varepsilon}\sum_{T\in\mathcal{T}} T.dbf(t)$). Hence, for small values of ε these are task sets which in a sense can "heavily" load the processor.

The above algorithms by themselves do not result in a polynomial time algorithm for approximate schedulability analysis if $t_{\max}$ is pseudo-polynomial in the size of the problem specification, since then a pseudo-polynomial number of checks have to be done. To avoid this, if there are m tasks in $\mathcal{T}$ then we instead perform a check only for $t = K, 2K, \ldots, (\lfloor \frac{t_{\max}}{K} \rfloor + 1)K$, where $K = \frac{\delta t_{\max}}{poly(m)}$. Here δ is an input error parameter to the algorithm (similar to ε in the case of approximating the demand-bound function) and $poly(m)$ is any polynomial function of m. Hence, the total number of checks is now $O(\frac{poly(m)}{\delta})$, which is polynomial in the size of the input.

We now bound the error incurred by such an algorithm. Consider the algorithm where we check the value of $\sum_{T\in\mathcal{T}} T.dbf(t)$ at $t = K, 2K, \ldots, (\lfloor \frac{t_{\max}}{K} \rfloor + 1)K$ and return `NOT SCHEDULABLE` if for any of these values of t, $\sum_{T\in\mathcal{T}} T.dbf(t) > t$ and else we return `SCHEDULABLE`. If $\mathcal{T}$ is schedulable, then clearly such an algorithm always returns the correct answer. But for some task sets which are not schedulable, the algorithm might incorrectly return `SCHEDULABLE` as well. However, in such cases a job from such a task set might miss its deadline by at most K time units.

The idea behind the algorithm is that since $\sum_{T\in\mathcal{T}} T.dbf(t)$ is a non-decreasing function of t, if its value at some t' exceeds t' by a large number then even with a polynomial number of checks it would be possible to come across some t close to t' at which the value of the function exceeds t. To see this, consider some time interval $[iK, (i+1)K]$. If $\sum_{T\in\mathcal{T}} T.dbf(iK) \leq iK$ and $\sum_{T\in\mathcal{T}} T.dbf((i+1)K) = (i+1)K$ (and hence the condition for schedulability is met at these two points), then in the worst case $\sum_{T \in \mathcal{T}} T.dbf(iK + \Delta) = (i+1)K$, where $\Delta \to 0$, and hence a job from $\mathcal{T}$ can miss its deadline by at most K time units. This argument applies to all intervals, the end points of which are only checked by the algorithm.

As in the case of the previous approximate schedulability analysis based on approximating the demand-bound function, here, too, it is possible to design a pessimistic algorithm. For this we check the value of $\sum_{T \in \mathcal{T}} T.dbf(t)$ at $t = K, 2K, \ldots, (\lfloor \frac{t_{\max}}{K} \rfloor + 1)K$ and return `NOT SCHEDULABLE` if for any of these values of t, $\sum_{T \in \mathcal{T}} T.dbf(t) > t - K$ and else we return `SCHEDULABLE`.

If $\mathcal{T}$ is not schedulable then this algorithm always returns the correct answer. To see this, suppose that $\mathcal{T}$ is not schedulable. Then there exists an interval $[iK, (i+1)K]$ such that for some $t \in [iK, (i+1)K]$, $\sum_{T \in \mathcal{T}} T.dbf(t) > t$. Since $\sum_{T \in \mathcal{T}} T.dbf(t)$ is a non-decreasing function, $\sum_{T \in \mathcal{T}} T.dbf((i+1)K) > iK$ and hence our algorithm returns `NOT SCHEDULABLE`. For task sets $\mathcal{T}$ which are schedulable, this algorithm can return an incorrect answer and clearly the error in these cases is bounded by K, i.e. for such $\mathcal{T}$ there exist time intervals of length t over which the processor can be occupied for at least $t - K$ time units. Since $K = \frac{\delta t_{\max}}{poly(m)}$, a smaller value of δ reduces the maximum error that can be incurred by both the optimistic and the pessimistic algorithms, at the cost of increasing the number of checks to be performed and hence the running time of the algorithm.

3.7.1.2 Bounding the total error

Now we present the algorithms obtained by combining the two steps described above (i.e. approximating the demand-bound function, and performing a polynomial instead of a pseudo-polynomial number of checks to verify whether the sum of the demand-bound functions for any t exceeds t). As before, we present two classes of algorithms—optimistic and pessimistic—and give a bound on the maximum error that can be incurred in both the cases. Both of these algorithms perform an approximate schedulability analysis in polynomial time, and the total error incurred depends on the values of the input error parameters ε and δ. The smaller these values, the less is the error but at the cost of the running time increasing appropriately.

Optimistic algorithms: For algorithms of this class, we check the value of $\sum_{T \in \mathcal{T}} T.dbf'(t)$ at $t = K, 2K, \ldots, (\lfloor \frac{t_{\max}}{K} \rfloor + 1)K$ and return `NOT SCHEDULABLE` if for any of these values of t, $\sum_{T \in \mathcal{T}} T.dbf'(t) > t$ and else return `SCHEDULABLE`. As before, $T.dbf'(t)$ is an approximate value of $T.dbf(t)$ such that $T.dbf(t) \geq T.dbf'(t) \geq f(\varepsilon)T.dbf(t)$ for any t. Note that this algorithm takes as input two error parameters ε and δ.

If $\mathcal{T}$ is schedulable then $\sum_{T \in \mathcal{T}} T.dbf(t)$ is less than or equal to t for all values of t at which this sum is checked, and since $T.dbf(t) \geq T.dbf'(t)$, our algorithm is guaranteed to return `SCHEDULABLE`.

Now consider the case where $\mathcal{T}$ is not schedulable. Here our algorithm can return an incorrect answer, and there are two sources of possible error. Consider any t at which the value of $\sum_{T\in\mathcal{T}} T.dbf'(t)$ is checked by our algorithm. If for any such t, $\sum_{T\in\mathcal{T}} T.dbf(t) > t$ but $\sum_{T\in\mathcal{T}} T.dbf'(t) \leq t$ then we incur a maximum error of $(1 - f(\varepsilon)) \sum_{T\in\mathcal{T}} T.dbf(t)$ which is less than or equal to $\frac{1-f(\varepsilon)}{f(\varepsilon)} \sum_{T\in\mathcal{T}} T.dbf'(t)$.

Secondly, consider any interval $[iK, (i+1)K]$ such that the value of $\sum_{T\in\mathcal{T}} T.dbf'(t)$ is checked at $t = iK$ and $t = (i+1)K$ and for both these values of t, the sum is less than or equal to t. The worst case error incurred in such a case occurs when $\sum_{T\in\mathcal{T}} T.dbf'(iK + \Delta) = (i+1)K$ where $\Delta \to 0$, and therefore this error is equal to K. Taking into account that $\sum_{T\in\mathcal{T}} T.dbf(iK) \leq \frac{1}{f(\varepsilon)} \sum_{T\in\mathcal{T}} T.dbf'(iK)$, the total error incurred by the algorithm within this interval is equal to $K + \frac{1}{f(\varepsilon)} \sum_{T\in\mathcal{T}} T.dbf'(iK) - \sum_{T\in\mathcal{T}} T.dbf(iK) \leq K + \frac{1-f(\varepsilon)}{f(\varepsilon)} \sum_{T\in\mathcal{T}} T.dbf'(iK)$.

Since for $t = K, 2K, \ldots, (\lfloor \frac{t_{\max}}{K} \rfloor + 1)K$, the value of $T.dbf'(t)$ is maximized at $t = (\lfloor \frac{t_{\max}}{K} \rfloor + 1)K$, the maximum possible total error incurred by the algorithm is equal to

$$K + \frac{1-f(\varepsilon)}{f(\varepsilon)} \sum_{T\in\mathcal{T}} T.dbf'((\lfloor \frac{t_{\max}}{K} \rfloor + 1)K)$$

Pessimistic algorithms: Here we check the value of $\sum_{T\in\mathcal{T}} \frac{1}{f(\varepsilon)} T.dbf'(t)$ at $t = K, 2K, \ldots, (\lfloor \frac{t_{\max}}{K} \rfloor + 1)K$ and return `NOT SCHEDULABLE` if for any of these values of t, $\frac{1}{f(\varepsilon)} \sum_{T\in\mathcal{T}} T.dbf'(t) > t - K$. Clearly, this algorithm is guaranteed to return the correct answer if $\mathcal{T}$ is not schedulable. But it might err if $\mathcal{T}$ is schedulable. To bound the error in such cases, suppose that for some iK, $1 \leq i \leq (\lfloor \frac{t_{\max}}{K} \rfloor + 1)$, $\frac{1}{f(\varepsilon)} \sum_{T\in\mathcal{T}} T.dbf'(iK) > (i-1)K$ (and therefore our algorithm returns `NOT SCHEDULABLE`) but $\sum_{T\in\mathcal{T}} T.dbf(iK) \leq iK$. Hence, the error incurred at iK is equal to

$$\begin{aligned}
& K + \frac{1}{f(\varepsilon)} \sum_{T\in\mathcal{T}} T.dbf'(iK) - \sum_{T\in\mathcal{T}} T.dbf(iK) \\
\leq\ & K + \frac{1}{f(\varepsilon)} \sum_{T\in\mathcal{T}} T.dbf'(iK) - \sum_{T\in\mathcal{T}} T.dbf'(iK) \\
=\ & K + \frac{1-f(\varepsilon)}{f(\varepsilon)} \sum_{T\in\mathcal{T}} T.dbf'(iK)
\end{aligned}$$

Hence, the maximum error incurred by this algorithm is also equal to

$$K + \frac{1-f(\varepsilon)}{f(\varepsilon)} \sum_{T\in\mathcal{T}} T.dbf'((\lfloor \frac{t_{\max}}{K} \rfloor + 1)K)$$

The running times of both the optimistic and the pessimistic algorithms are polynomial, assuming that there exists a polynomial time approximation algorithm for computing $T.dbf(t)$ for any task T and time interval of length

t. For example, if the later algorithm is an FPTAS, and the specification of any task is of the size $O(n)$ and $\mathcal{T}$ contains m such tasks, then the total running time of any of the algorithms for approximate schedulability analysis is $O(poly(m, n, \frac{1}{\delta}, \frac{1}{\varepsilon}))$.

3.7.2 Algorithms for approximate schedulability analysis

Now we apply the results obtained in Section 3.7.1.1 to our task model described in Section 3.2, and derive algorithms for approximate schedulability analysis for this model. Towards this, we first show that there exists a fully-polynomial time approximation scheme (FPTAS) for computing the demand-bound function $T.dbf(t)$ for any time interval of length t and for any task graph T in our task model. We then combine this algorithm with the scheme for checking the sum of the demand-bound functions only for polynomially bounded different values of t. Finally, we show that the resulting algorithm runs in polynomial time and derive the maximum error values incurred for the optimistic and the pessimistic versions of this algorithm.

3.7.2.1 Approximating the *demand-bound function*

The algorithms given in this section constitute the *Building Block 1* shown in Figure 14. First, let us only consider task graphs in our restricted model given in Section 3.5, i.e. the control flows from the source to the sink vertex only once and there is no recurring execution of the task graphs. Now, consider Algorithm 1 in Section 3.5.1, for computing the value of the demand-bound function $T.dbf(t)$ for any task graph T in this restricted model. As in Section 3.5.1, we assume that a given task graph T has n vertices denoted by $v_1, \ldots, v_n$ and there can be a directed edge from v_i to v_j only if $i < j$. Further, associated with each vertex v_i is its execution requirement $e(v_i)$ which is assumed to be integral, and its deadline $d(v_i)$. Associated with each edge (v_i, v_j) is the minimum intertriggering separation $p(v_i, v_j)$. Given Algorithm 1, any $t \geq 0$, and an $0 < \varepsilon \leq 1$, let T_t be the subgraph of T consisting only of those vertices v_i for which $d(v_i) \leq t$, and let E_t denote the maximum execution requirement of a vertex from among all vertices of T_t. Now we scale all the execution requirements associated with the vertices of T_t by $K = \varepsilon E_t/n$ i.e. $e'(v_i) = \lfloor e(v_i)/K \rfloor$ and run the algorithm with the new $e'(v_i)$s and the graph T_t. Let V be the set of vertices (with the scaled execution requirements) that result in the computation of $T.dbf(t)$ in this algorithm. We claim that the summation of the original (unscaled) execution requirements of these vertices is greater than or equal to $(1-\varepsilon)$ times the actual demand-bound function for the task graph for this value of t. Further, since this algorithm now runs in time $O(n^4/\varepsilon)$, (with the scaled execution requirements), it is an FPTAS for computing $T.dbf(t)$. We denote this approximate value of $T.dbf(t)$ computed by this algorithm by $T.dbf'(t)$.

Thm. 12: *There exists a fully polynomial-time approximation scheme for computing $T.dbf(t)$. For any ε the algorithm runs in $O(n^4/\varepsilon)$ time, where n is the number of vertices in the task graph T.*

Proof: Given a task graph T with n vertices and any time interval t, consider the subgraph of T which consists of only those vertices v_i for which $d(v_i) \geq t$. Let $E = \max_i e(v_i)$ among these nodes. Clearly, $T.dbf(t) \geq E$. For any $0 < \varepsilon \leq 1$, let $K = \varepsilon E/n$. Now scale the execution requirements of all the vertices of this subgraph as follows: $e'(v_i) = \lfloor e(v_i)/K \rfloor$. Then clearly,

$$\frac{e(v_i)}{K} - 1 \leq e'(v_i) \leq \frac{e(v_i)}{K}$$

This implies that

$$\begin{aligned} e(v_i) &\geq Ke'(v_i) && (3.7) \\ Ke'(v_i) &\geq e(v_i) - K && (3.8) \end{aligned}$$

We run the dynamic programming algorithm (Algorithm 1) with the scaled execution requirements $e'(v_i)$ on this subgraph. Let some path $\pi = v_1, \ldots, v_k$ be the output of the dynamic programming algorithm (which results in the computation of $T.dbf(t)$ by Algorithm 1). Let π_{OPT} be the path in the task graph T which results in the computation of the exact $T.dbf(t)$. Then,

$$\begin{aligned} \sum_{v \in \pi} e(v) &\geq K \sum_{v \in \pi} e'(v) \quad \text{(from Inequality (3.7))} \\ &\geq K \sum_{v \in \pi_{OPT}} e'(v) \quad \text{(since } \pi \text{ is optimal with the } e'(v)\text{s)} \\ &\geq \sum_{v \in \pi_{OPT}} (e(v) - K) \quad \text{(from Inequality (3.8))} \\ &= \sum_{v \in \pi_{OPT}} e(v) - K|\pi_{OPT}| \\ &\geq \sum_{v \in \pi_{OPT}} e(v) - Kn = \sum_{v \in \pi_{OPT}} e(v) - \varepsilon E \\ &\geq T.dbf(t) - \varepsilon T.dbf(t) = (1-\varepsilon) T.dbf(t) \end{aligned}$$

Therefore, if we denote the sum $\sum_{v \in \pi} e(v)$ by $T.dbf'(t)$ then $T.dbf(t) \geq T.dbf'(t) \geq (1-\varepsilon)T.dbf(t)$. The running time of Algorithm 1 with the unscaled execution requirements is $O(n^3 E)$. But, since the maximum execution requirement E is now n/ε, the running time of this fully polynomial-time approximation scheme is $O(n^4/\varepsilon)$. □

Approximate schedulability analysis for the restricted task model: We now show that using this approximate value of the demand-bound function $T.dbf'(t)$, it is possible to obtain polynomial time approximate decision algorithms for testing schedulability in our restricted task model. Note that in this restricted model, the *Building Block 2* of Figure 14 is not required, since as we show below, the number of checks of the sum of the demand-bound functions can be polynomially bounded if an approximate value of the demand-bound function (i.e. $T.dbf'(t)$) is used. First let us consider the preemptive case, for which already gave a pseudo-polynomial time algorithm in Section 3.5.1. To obtain the approximate decision algorithm, note that for all $t \geq 0$, there can be at most n distinct values of E_t for any task graph. For each such E_t, we consider the corresponding subgraph that gives rise to this E_t as described above, and scale the execution requirements of the vertices of this subgraph by $K = \varepsilon E_t / n$. In each such subgraph T_t, the number of values of time intervals t' at which the value of $T_t.dbf'(t')$ changes is bounded by $O(n^2/\varepsilon)$, and hence the number of values of time intervals t at which the value of $\sum_{T \in \mathcal{T}} T.dbf'(t)$ changes is bounded by $O(|\mathcal{T}|n^3/\varepsilon)$. The approximate decision algorithm for dynamic-priority preemptive schedulability analysis is now given as Algorithm 4.

Algorithm 4 Approximate decision algorithm for schedulability analysis

```
Require: Task set $\mathcal{T}$ and a real $0 < \varepsilon \leq 1$
  $decision \leftarrow$ SCHEDULABLE
  for all values of $t$ at which $T.dbf'(t)$ changes for any $T \in \mathcal{T}$ do
    if $\frac{1}{1-\varepsilon} \sum_{T \in \mathcal{T}} T.dbf'(t) > t$ then {Condition (*)}
      $decision \leftarrow$ NOT SCHEDULABLE
    end if
  end for
  return $decision$
```

Thm. 13: *If a task set $\mathcal{T}$ is not schedulable then Algorithm 4 always returns the correct answer. If $\mathcal{T}$ is schedulable and $t \geq \frac{1}{1-\varepsilon} \sum_{T \in \mathcal{T}} T.dbf'(t)$ for all values of t, then the algorithm always returns the correct answer* SCHEDULABLE*, otherwise it might return* NOT SCHEDULABLE. SCHEDULABLE *answers are always correct. The running time of the algorithm is $O(|\mathcal{T}|^2 n^5 \varepsilon^{-2} \log n)$, if all task graphs have $O(n)$ vertices.*

For each task T, computing the $t_{n,e}$ (following the same notation as used in Section 3.5.1) values for each of its subgraphs T_t, using Algorithm 1 and the scaled execution requirements requires $O(n^4/\varepsilon)$ time, and these values are stored in a table. Hence computing all such values for all the task graphs in $\mathcal{T}$ takes $O(n^5|\mathcal{T}|/\varepsilon)$ time. For each value of t for which $\sum_{T \in \mathcal{T}} T.dbf'(t)$ changes, computing $T.dbf'(t)$ for any $T \in \mathcal{T}$ requires a binary search to identify the appropriate table corresponding to a subgraph T_t, and then a linear search through the table. Therefore, computing the value of $\sum_{T \in \mathcal{T}} T.dbf'(t)$ for any value of t takes $O(|\mathcal{T}|n^2\varepsilon^{-1} \log n)$ time. Hence the total running time of Algorithm 4 is $O(|\mathcal{T}|^2 n^5 \varepsilon^{-2} \log n)$. The algorithm is overly pessimistic in the sense that for certain schedulable task sets it might return NOT SCHEDULABLE. However, for task sets which can be in some sense comfortably scheduled even in the worst case, leaving some idle processor time (which can be parameterized by ε), the algorithm always returns SCHEDULABLE. Therefore, any ε characterizes a class of task sets for which the algorithm errs. As described in Section 3.7.1.1, decreasing ε reduces this class of such task sets for which the algorithm errs, at the cost of increasing the running time quadratically in $1/\varepsilon$, thereby giving a fully polynomial-time approximate decision scheme for approximate schedulability analysis.

It may be noted that changing Condition $(*)$ in Algorithm 4 to

```
**if** $\sum_{T \in \mathcal{T}} T.dbf'(t) > t$ **then**
    $decision \leftarrow$ NOT SCHEDULABLE
**end if**
```

will result in an overly optimistic algorithm which might incorrectly return SCHEDULABLE for certain classes of not schedulable task sets. For all schedulable task sets it always returns SCHEDULABLE, and NOT SCHEDULABLE answers are always correct. The task sets for which the algorithm might err are those in which the cumulative execution requirement by tasks of $\mathcal{T}$ within any time interval of length t exceeds the maximum execution requirement that can be feasibly scheduled, by an amount of less than $\varepsilon \sum_{T \in \mathcal{T}} T.dbf(t)$ time units. Again, decreasing ε reduces the class of such task sets, at the cost of the running time increasing linearly in $1/\varepsilon$.

In the non-preemptive case, i.e. in Algorithm 3, we can follow the same arguments as in the preemptive case and design approximate decision algorithms for testing schedulability using the approximate values $T.dbf'(t)$ and $T.dbf'^{v}(t)$. The only difference compared to the preemptive case, is that the running time of this algorithm is different, and this is derived below.

Following the same notation as in the preemptive case, in each subgraph T_t of T, the number of values of time intervals t' at which the value of $T_t.dbf'(t')$ changes is also bounded by $O(n^2/\varepsilon)$, and hence the number of values of time intervals t at which the value of $\sum_{T \in \mathcal{T}} T.dbf'(t)$ changes is bounded by $O(|\mathcal{T}|n^3/\varepsilon)$.

It follows that in Step 2 of Algorithm 3, it is sufficient to run the loop only for $O(|\mathcal{T}|n^3/\varepsilon)$ values of $\hat{\tau}$, since the value of $\sum_{T \in \mathcal{T}} T.dbf'(\hat{\tau})$ can change at most these many number of times. Therefore, the loop in Step 2 executes for a total of $O(|\mathcal{T}|^2 n^4/\varepsilon)$ times.

Again, as in the preemptive case, computing all $t_{n,e}$ values for all the task graphs in $\mathcal{T}$ takes $O(n^5|\mathcal{T}|/\varepsilon)$ time. The Step 25 in Algorithm 3 dominates the running time among all the steps inside the loop (Steps 3-27), and requires a computation of $T_i.dbf'^{v}(t) + \sum_{T \in \hat{\mathcal{T}}} T.dbf'(t + e_{max})$ where $t = \hat{\tau} + d(v)$. As in the preemptive case, computing $T.dbf'(t)$ for any $T \in \mathcal{T}$ requires a binary search to identify the appropriate table corresponding to a subgraph T_t, and then a linear search through this table. Therefore, this requires $O(n^2\varepsilon^{-1}\log n)$ time. The exactly same time is required for computing $T.dbf'^{v}(t)$. Hence, computing the value of $T_i.dbf'^{v}(t) + \sum_{T \in \hat{\mathcal{T}}} T.dbf'(t + e_{max})$ for $t = \hat{\tau} + d(v)$ in Step 25 requires a total of $O(|\mathcal{T}|n^2\varepsilon^{-1}\log n)$ time. Therefore, the total run time of Algorithm 3 using approximate values of the demand-bound function is $O(|\mathcal{T}|^3 n^6 \varepsilon^{-2}\log n)$.

The general task model: Now we consider general task graphs, in which the recurring behaviour in their execution is taken into account. For these task graphs, we give a scheme for computing an approximate value of the demand-bound function $T.dbf(t)$ for any t. First, let us consider the problem of computing an approximate value of $T.dbf(t)$ for a "small" value of t, where *small* has the same meaning as described in Section 3.4.3. Towards this, we take two copies of a task graph T and join them as in Section 3.4.3.1. Let us call this new task graph T'. If T has n vertices, then T' has $2n$ vertices. Now, using Algorithm 1 with scaled-down execution requirements of the vertices of T' (as described in the begining of this section), it may be seen that for any ε, the algorithm outputs $T'.dbf'(t)$ in $O(n^4/\varepsilon)$ time. Moreover, as in the case of the restricted task model, $T'.dbf'(t)$ for any t and ε, is greater than or equal to $(1-\varepsilon)$ times the value of $T'.dbf(t)$.

Since $E(T)$ (defined in Section 3.4.1) for any task graph T with n vertices can be computed in $O(n^2)$ time, using the above scheme for computing $T'.dbf'(t)$ for small values of t along with Equation 3.2, the demand-bound function $T.dbf(t')$ for a task graph T can be approximated for any general t' in $O(n^4/\varepsilon)$ time. As before, we denote this approximate value of $T.dbf(t')$ by $T.dbf'(t')$, and the following inequality holds for any t' and ε: $T.dbf(t') \geq T.dbf'(t') \geq (1-\varepsilon)T.dbf(t')$.

3.7.2.2 Checking the sum of the *demand-bound functions*

This section describes the *Building Block 2* of Figure 14 in the context of our task model. Recall from Section 3.4.1 that the exact algorithm for schedulability analysis for this model requires a pseudo-polynomial number of checks of the sum of the demand-bound functions. Following our framework for approximate schedulability analysis described in Section 3.7.1.1, we now show that combined with the approximate demand-bound function computed in the last section, a polynomial number of checks result in a bounded error. As in Section 3.7.1.1, we present an optimistic and a pessimistic algorithm, and in addition present a third algorithm where both SCHEDULABLE and NOT SCHEDULABLE answers can be wrong. However, we show that the worst case error incurred in either of these wrong decisions is less than the error incurred by the previous two algorithms.

The bounds on the error we obtain here for all the algorithms are tighter than the bounds derived in Section 3.7.1.1 for the abstract model. Given the FPTAS described in the last section for approximating the value of $T.dbf(t)$, it is possible to obtain the following bounds on the approximate value of the demand-bound function $T.dbf'(t)$ (see proof of Theorem 12).

$$T.dbf'(t) + \varepsilon E_T \geq T.dbf(t)$$

where E_T is the maximum execution requirement of any vertex in the task graph T. Further, since $\frac{1}{1-\varepsilon}T.dbf'(t) \geq T.dbf(t)$, we have

$$T.dbf(t) \leq \min\{\frac{1}{1-\varepsilon}T.dbf'(t), T.dbf'(t) + \varepsilon E_T\} \tag{3.9}$$

Algorithm 5 Optimistic algorithm $(t_{\max}, \delta)$

```
Require: $t_{\max}, \delta$, $m$ = number of task graphs in $\mathcal{T}$, for each task graph $T$, $E_T$
    is the maximum execution requirement of any vertex in $T$
  $K \leftarrow \frac{\delta t_{\max}}{poly(m)}$
  $error \leftarrow 0$
  $decision \leftarrow$ SCHEDULABLE
  for $t \leftarrow 1$ to $\lfloor \frac{t_{\max}}{K} \rfloor + 1$ do
    if $\sum_{T \in \mathcal{T}} T.dbf'(tK) > tK$ then
      $decision \leftarrow$ NOT SCHEDULABLE
    else
      $error\_in\_this\_interval \leftarrow \max\{\min\{\frac{1}{1-\varepsilon} \sum_{T \in \mathcal{T}} T.dbf'(tK),$
                          $\sum_{T \in \mathcal{T}} T.dbf'(tK) + \varepsilon \sum_{T \in \mathcal{T}} E_T\} - (t-1)K, 0\}$
      $error \leftarrow \max\{error, error\_in\_this\_interval\}$
    end if
  end for
  return($decision, error$)
```

which gives the better of the two bounds on $T.dbf(t)$ for any value of t.

For any task set $\mathcal{T}$, Algorithm 5 always returns the correct answer if $\mathcal{T}$ is schedulable but might err if $\mathcal{T}$ is not schedulable. Hence, whenever this algorithm returns NOT SCHEDULABLE, the decision is guaranteed to be correct. But SCHEDULABLE answers might be wrong. From Section 3.7.1.2, we obtain that the maximum possible error that can be incurred by the algorithm is equal to $K + \frac{\varepsilon}{1-\varepsilon} \sum_{T \in \mathcal{T}} T.dbf'((\lfloor \frac{t_{\max}}{K} \rfloor + 1)K)$. However, using Inequality (3.9) we can obtain a tighter bound on the error, which is given by

$$K + \min\{\frac{\varepsilon}{1-\varepsilon} \sum_{T \in \mathcal{T}} T.dbf'((\lfloor \frac{t_{\max}}{K} \rfloor + 1)K), \varepsilon \sum_{T \in \mathcal{T}} E_T\}$$

Clearly, for any given problem instance, the maximum error that is incurred will be lower than this theoretical worst case bound. Algorithm 5 computes this theoretical worst case error for each problem instance. To understand this, consider any interval $[iK, (i+1)K]$ such that $\sum_{T \in \mathcal{T}} T.dbf'(t)$ is checked at $t = iK$ and $t = (i+1)K$. If at $t = (i+1)K$, the computed upper bound on $\sum_{T \in \mathcal{T}} T.dbf(t)$ (computed using inequality (3.9)) is less than or equal to iK then the error incurred in the interval $(iK, (i+1)K]$ is equal to 0, since then for any $t \in (iK, (i+1)K]$, $\sum_{T \in \mathcal{T}} T.dbf(t)$ is guaranteed to be less than or equal to t. Alternatively, if the computed upper bound on $\sum_{T \in \mathcal{T}} T.dbf(t)$ is greater than iK then the maximum possible error in the interval $[iK, (i+1)K]$ is equal to the difference between this bound and iK. Algorithm 5 computes this error at each interval and outputs the maximum among the computed error values.

Algorithm 6 states the corresponding pessimistic algorithm in which SCHEDULABLE answers are guaranteed to be correct and NOT SCHEDULABLE answers might be wrong, and the maximum error is the same as in the case of Algorithm 5. Finally, Algorithm 7 uses the upper

Algorithm 6 Pessimistic algorithm ($t_{\max}, \delta$)

```
Require: $t_{\max}, \delta$, $m$ = number of task graphs in $\mathcal{T}$, for each task graph $T$, $E_T$ is the maximum execution requirement of any vertex in $T$ and $d_T$ is the minimum deadline associated with any vertex in $T$
  $K \leftarrow \frac{\delta t_{\max}}{poly(m)}$
  $decision \leftarrow$ SCHEDULABLE
  for $t \leftarrow 1$ to $\lfloor \frac{t_{\max}}{K} \rfloor + 1$ do
    if $\min\{\frac{1}{1-\varepsilon} \sum_{T \in \mathcal{T}} T.dbf'(d_T + tK), \sum_{T \in \mathcal{T}} T.dbf'(d_T + tK) + \varepsilon \sum_{T \in \mathcal{T}} E_T\} > d_T + (t-1)K$ then
      $decision \leftarrow$ NOT SCHEDULABLE
    end if
  end for
  return($decision$)
```

Algorithm 7 Algorithm with double sided error

```
Require: $t_{\max}, \delta, m$ = number of task graphs in $\mathcal{T}$, for each task graph $T$, $E_T$ is the maximum execution requirement of any vertex in $T$
  $K \leftarrow \frac{\delta t_{\max}}{poly(m)}$
  $decision \leftarrow$ SCHEDULABLE
  for $t \leftarrow 1$ to $\lfloor \frac{t_{\max}}{K} \rfloor + 1$ do
    if $\min\{\frac{1}{1-\varepsilon} \sum_{T \in \mathcal{T}} T.dbf'(tK), \sum_{T \in \mathcal{T}} T.dbf'(tK) + \varepsilon \sum_{T \in \mathcal{T}} E_T\} > tK$ then
      $decision \leftarrow$ NOT SCHEDULABLE
    end if
  end for
  return($decision$)
```

bound on the value of $T.dbf(t)$ as given by Inequality (3.9), along with the optimistic version of the checking procedure given by our *Building Block 2* in Figure 14. This algorithm incurs a double sided error—SCHEDULABLE answers can be wrong, but the maximum error in this case is bounded by K. NOT SCHEDULABLE answers can be wrong too, but the error in this case is bounded by $\min\{\frac{\varepsilon}{1-\varepsilon} \sum_{T \in \mathcal{T}} T.dbf'((\lfloor \frac{t_{\max}}{K} \rfloor + 1)K), \varepsilon \sum_{T \in \mathcal{T}} E_T\}$. Hence, the maximum error in either case is smaller than the maximum error incurred by the optimistic and the pessimistic algorithms.

Running Times: Following the same reasoning as in Section 3.7.2.1 for the restricted task model, it can be shown that computing all possible values of $T.dbf'(t)$ for *small* values of t and for all task graphs $T \in \mathcal{T}$, takes $O(n^5 m/\varepsilon)$ time, where each task graph in $\mathcal{T}$ contains $O(n)$ vertices and $\mathcal{T}$ contains m task graphs. All of these values are first computed and stored in a table. During the second phase of the algorithm (when the sum of the demand-bound functions are checked to determine if they exceed t), for any t, computing the value of $\sum_{T \in \mathcal{T}} T.dbf'(t)$ needs a table lookup which takes $O(n^2 m \varepsilon^{-1} \log n)$

time. Since the sum of the demand bound functions is checked for $O(\frac{poly(m)}{\delta})$ values of t, the running time of any of the three algorithms described above is $O(n^5 m\varepsilon^{-1} + n^2 m\varepsilon^{-1}\delta^{-1} poly(m) \log n)$. Hence, the approximate schedulability analysis algorithm for our task model runs in polynomial time.

3.7.3 Other task models

Our results in Section 3.7.1.1 also imply polynomial time algorithms for approximate schedulability analysis for models such as sporadic, multiframe and the generalized multiframe. All of these models can be considered as special cases of the recurring real-time task model (for which, the results of Section 3.7.2.1 and 3.7.2.2 directly hold). Nevertheless, we especially point them out here since the resulting algorithms for approximate schedulability analysis are considerably simpler compared to those for the recurring real-time task model. The reason for this is that in all of these models the demand-bound function for any t can be computed (exactly) in polynomial time. Therefore, the pseudo-polynomial running times of the known algorithms for schedulability analysis for all of these models are attributed to the pseudo-polynomial number of tests to verify if for any t the sum of the demand-bound functions exceeds t.

If $t_{\max}$ is the number of tests to be performed, then as shown in Section 3.7.1.1, an algorithm for approximate schedulability analysis for any of these models performs only $O(\frac{t_{\max}}{K})$ tests where K, as before, is equal to $\frac{\delta t_{\max}}{poly(|\mathcal{T}|)}$ (δ is the input error parameter to the algorithm). The error, in the case of both optimistic and pessimistic algorithms is bounded by $K = \frac{\delta t_{\max}}{poly(|\mathcal{T}|)}$. Since the demand-bound function for any t in these models can be computed in polynomial time, and the number of tests is bounded by $O(\frac{poly(|\mathcal{T}|)}{\delta})$, this implies a polynomial time algorithm for approximate schedulability analysis.

3.8 Experimental results

In this section we report some experimental results obtained by testing Algorithm 5 on a set of randomly generated task graphs. The parameters used for generating these graphs are motived by a typical packet processing scenario—we take into account the running times of common packet processing tasks on processing elements that are commonly found in any network packet processor.

In spite of the theoretical guarantees (both in terms of the running time, and the maximum error that can be incurred) associated with the algorithms presented in this chapter, these results are interesting because of two reasons. Firstly, many approximation schemes (and many of them for different scheduling problems) are difficult to implement and in practice might have running times which are comparable or even worse than the equivalent, simpler exponential time algorithms, for all practical input instances. It turns out that this is not the case for our problem. Secondly, there are two input *error parameters* in

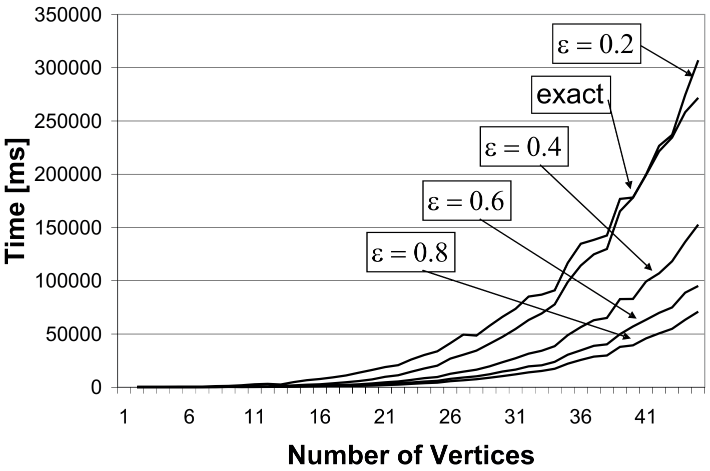

Fig. 15: Running times for computing the demand-bound function $T.dbf(t)$ for a single task graph with $E = 200$. The different values of the error parameter ε correspond to the respective approximation algorithms, and **exact** refers to the running time of the pseudo-polynomial time algorithm (Algorithm 1).

our algorithms—ε and δ—which represent a tradeoff between running time and the quality of the results obtained. Therefore, it is interesting to identify suitable combinations of these two parameters for any realistic input instance.

The results reported here are only for the preemptive dynamic-priority schedulability analysis algorithm. The algorithms for the non-preemptive case, and those for static-priority would give similar results both in terms of quality and running time, since they are based on similar underlying principles. However, it may be pointed out here, that it would be difficult to verify the quality of the results obtained in the case of the static-priority algorithms since only a sufficient condition for schedulability is known—unless of course the graphs are simple enough.

We have implemented an exact schedulability analysis algorithm, based on the dynamic programming algorithm (Algorithm 1) for computing the demand-bound function and testing if the sum of the demand-bound functions for all the task graphs exceed t for all $t \leq t_{\max}$. This algorithm runs in pseudo-polynomial time. We compare the running time of this algorithm against the algorithm for approximate schedulability analysis.

For our experiments, the synthetic task graphs were randomly generated using two parameters. The first is the maximum execution requirement associated with any vertex of the graph, which we denote by E. Both, the running time of the pseudo-polynomial exact algorithm and the quality of the results obtained by the approximate schedulability analysis depend on this parameter. We call the second parameter the *connectivity factor*. If $v_1, \ldots, v_n$ are the vertices of a task graph such that there is an edge from v_i to v_j only if $j > i$, then while gen-

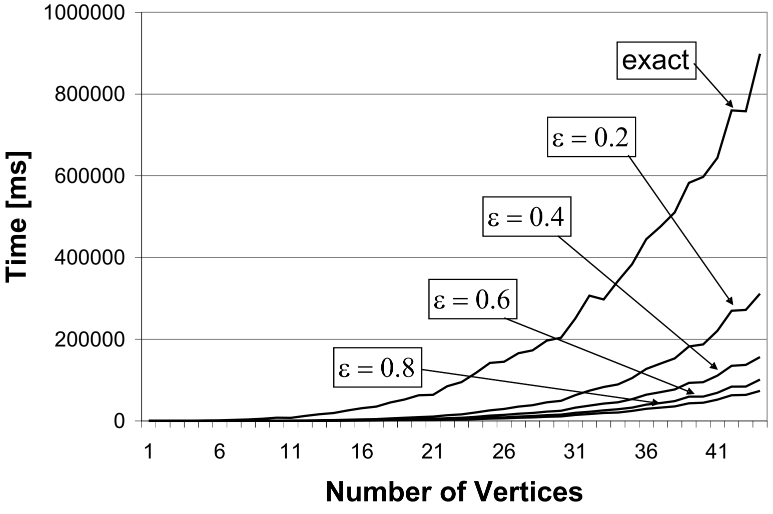

Fig. 16: Running times for computing the demand-bound function $T.dbf(t)$ for a single task graph with $E = 200$.

erating the graph, for each vertex v_j we construct an edge from v_i to v_j with a probability equal to the connectivity factor of the graph, for all $i = 1, \ldots, j-1$.

Figures 15 and 16 show the running times involved in computing the demand bound function for a single task graph using the exact pseudo-polynomial algorithm (i.e. Algorithm 1) and the FPTAS for four different values of ε, when the number of vertices in the graphs is gradually increased. In Figure 15, the maximum execution requirement of a vertex is set to 200, while in Figure 16 it is equal to 600.

The connectivity factor used for generating all the graphs was set to 0.4. To get an impression about the effect of the connectivity factor on the structure of a randomly generated graph, in Figure 17 we show some typical randomly generated graphs using different values of the connectivity factor. We decided to use a value of 0.4 because the generated graphs then have a realistic mixture of branches and consecutive tasks.

It may be noted from Figures 15 and 16 that the optimal choice of ε depends on E and the number of vertices in the task graph. For example in case of $E = 200$, the exact algorithm is better than the FPTAS for $\varepsilon = 0.2$ when the number of vertices in the graph increase beyond 40. To give an example of the values of E, that occur in a typical network packet processor, Table 3.8 states the approximate execution requirements of the different tasks involved in processing the voice flow (labelled as **Flow RT Send**) shown in Figure 6 of Chapter 2. Note that although these execution requirements are given in nanoseconds, were are of course only concerned with the absolute values (which we refer to as "time units"). Here we assume that RISC1 is a RISC processor like PowerPC, RISC2 is a processor like the ARM9TDMI, the μ-Engine is assumed to be similar to the ones present on the Intel IXP1200 and the DSP is like the

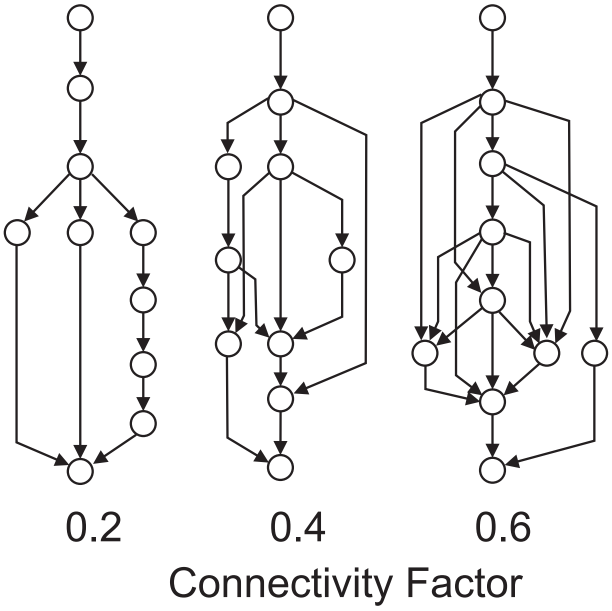

Fig. 17: Typical randomly generated task graphs for connectivity factors equal to $0.2, 0.4, 0.6$.

TMS320C620x. Following the values in this table, we believe that the choice of the maximum execution requirement associated with any vertex (E) set to a value around 600 is well motivated. The experiments with $E = 200$ have been shown here to illustrate the influence of the value of this parameter on the running time of the algorithms.

Figure 18 shows the exact value of the demand-bound function $T.dbf(t)$ computed by the pseudo-polynomial algorithm, and its upper and lower bounds ($T.dbf'(t) + \varepsilon E$ and $T.dbf'(e)$ respectively) computed by the approximation scheme. It should be noted that the value of $T.dbf'(t)$ for all values of t is almost equal to $T.dbf(t)$, and this is better than the worst case theoretical bound. The values shown in this graph are for a task graph with $E = 1000$, $\varepsilon = 0.6$ and the number of vertices in the task graph being equal to 30. Figure 19 shows the error incurred in approximating $T.dbf(t)$ for the same graph.

In Table 2 we show the running times involved in performing the test to determine if the sum of the demand-bound functions for any t, exceeds t (i.e. the *Building Block 2* in Figure 14). For the exact case, $t_{\max}$ tests are performed and in the case of approximate schedulability analysis, different values of the error parameter δ are used. Here the task set consists of three task graphs with 30 vertices per graph and the maximum execution requirement associate with any vertex is equal to 200, and $K = \frac{\delta t_{\max}}{m^6}$ (where $m = 3$ is the number of tasks in the task set). Note that the running times reported here do not involve the time required to compute the demand-bound functions. These are precomputed and stored in a table.

Lastly, in Figure 20 we show the percentage of wrong answers returned by the algorithm for different values of ε and δ. For this we have averaged the results obtained from a set of 600 task sets, each, as before, consisting of three

Task Name	RISC 1	RISC 2	μ-Engine	DSP
Voice Encoder	119460	119200	132200	11300
RTP Tx	100	110	110	160
UDP Tx	310	270	280	300
Build IP Header	130	130	110	190
Route Look Up	370	420	290	640
Calc Check Sum	200	180	150	110
ARP Look Up	300	330	230	500
Schedule	270	310	400	490

Tab. 1: Possible execution requirements (in ns) associated with the vertices involved in processing the voice flow, in the task graph shown in Figure 6 of Chapter 2.

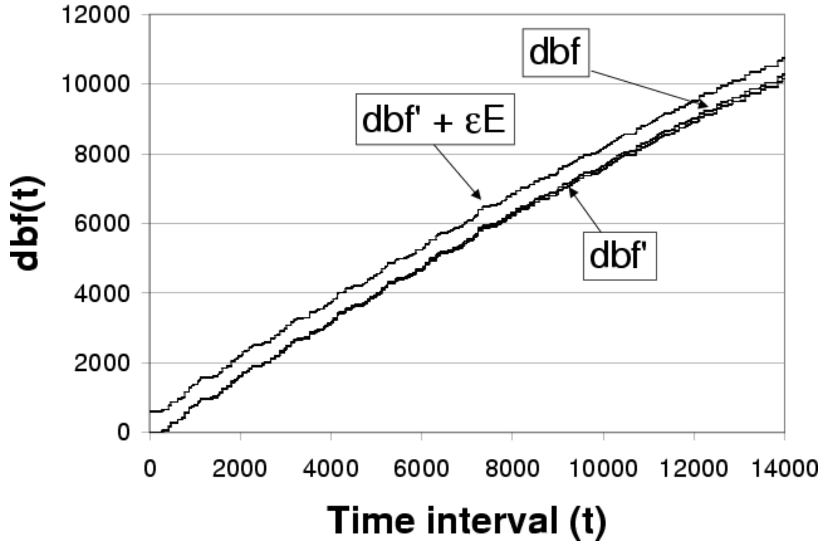

Fig. 18: The (exact values of the) demand-bound function $T.dbf(t)$ and the upper and lower bounds on its approximation, for different values of t.

		$\delta \rightarrow$				
		exact	**0.2**	**0.4**	**0.6**	**0.8**
	exact	775	36	18	11	9
ε	**0.2**	781	35	17	11	9
$\downarrow$	**0.4**	759	34	17	11	8
	0.6	749	34	16	11	8
	0.8	728	33	16	11	8

Tab. 2: Running times (in ms) of *Building Block 2* in Figure 14 for $K = \frac{\delta t_{\max}}{m^6}$

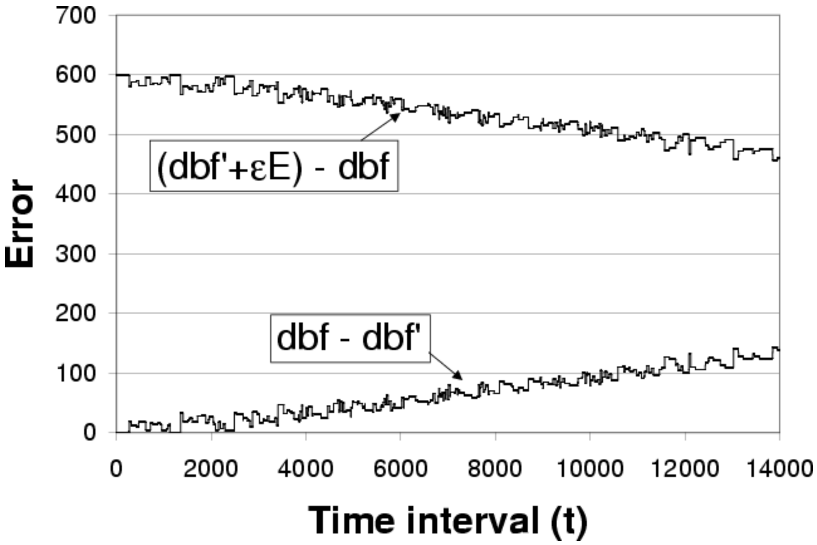

Fig. 19: The error incurred in approximating $T.dbf(t)$ for different values of t, for the example shown in Figure 18.

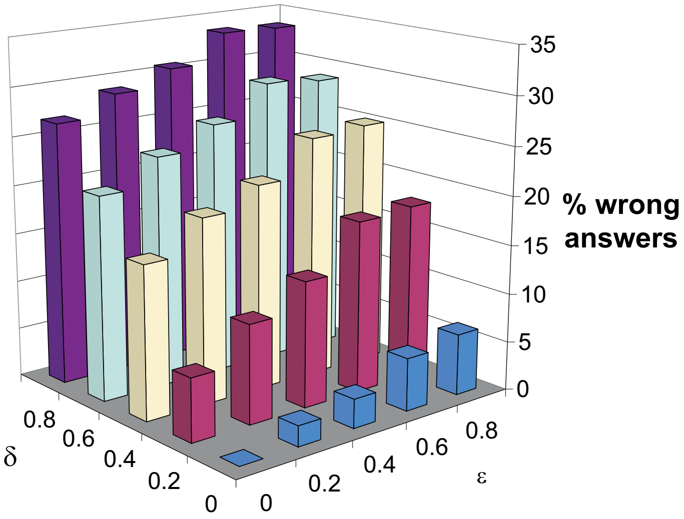

Fig. 20: Percentage of wrong answers averaged over 600 task sets, with $K = \frac{\delta t_{\max}}{m^4}$, for different values of ε and δ. The maximum execution requirement associated with any vertex of a task graph (E) is equal to 200, and the number of task graphs in any task set (m) is equal to 3.

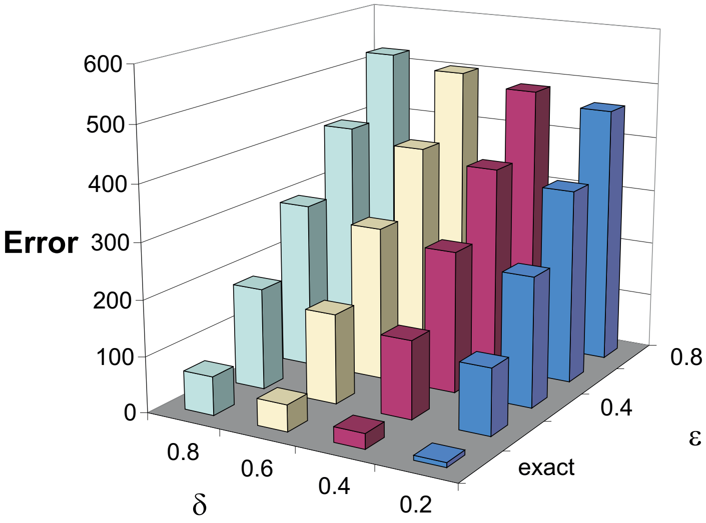

Fig. 21: Maximum error incurred by the decisions returned by the schedulability analysis algorithm, corresponding to the results shown in Figure 20.

task graphs with 30 vertices in each graph and maximum execution requirement of any vertex equal to 200. Here $K = \frac{\delta t_{\max}}{m^4}$ (where $m = 3$ is the number of tasks in a task set). Figure 20 shows the corresponding maximum *error* values obtained for different choices of ε and δ, again averaged over the 600 task sets.

Figures 22 and 23 show the results obtained with the same input but with K set to $\frac{\delta t_{\max}}{m^6}$. Note that as expected, because of the higher degree polynomial (m^6 instead of m^4), both the maximum error incurred and the percentage of wrong results decrease compared to the previous case. But now, more values of t are tested and hence the running time increases. All the task sets considered in the experiment were generated such that they either fully load the processor, or when not schedulable, the vertices miss their deadlines only by small amounts of time. Therefore, these represent most difficult cases for the approximate schedulability analysis algorithm. If the task sets are either "comfortably" schedulable, leaving the processor idle for a long intervals of time, or if they overload the processor in the sense that jobs miss their deadlines by long intervals of time, then the percentage of wrong results returned by the approximate schedulability analysis algorithm decrease. If the *load-factor* of a task set $\mathcal{T}$ is defined as $\max\{t = 1, \ldots, t_{\max} \mid \frac{\sum_{T \in \mathcal{T}} T.dbf(t)}{t}\}$, then the load-factor of all the task sets considered in our experiments were between 0.93 and 1.11. Lastly, among the 600 task sets considered around 50% were schedulable and the rest not.

All the CPU times reported here were measured on a a moderately loaded Sunblade 1000 running SunOS 5.8 with 750 MHz CPU and 2 GB RAM, and all the algorithms were implemented in Java.

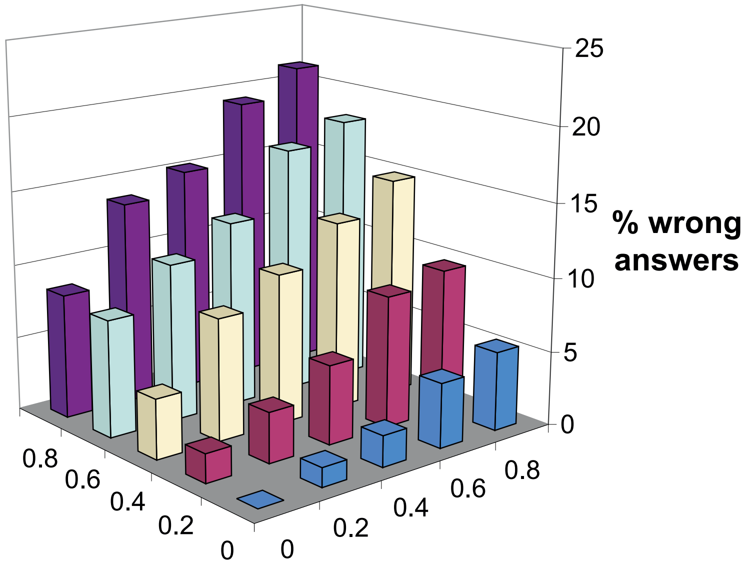

Fig. 22: Percentage of wrong answers averaged over 600 task sets, with $K = \frac{\delta t_{\max}}{m^6}$, for different values of ε and δ. The maximum execution requirement associated with any vertex of a task graph (E) is equal to 200, and the number of task graphs in any task set (m) is equal to 3.

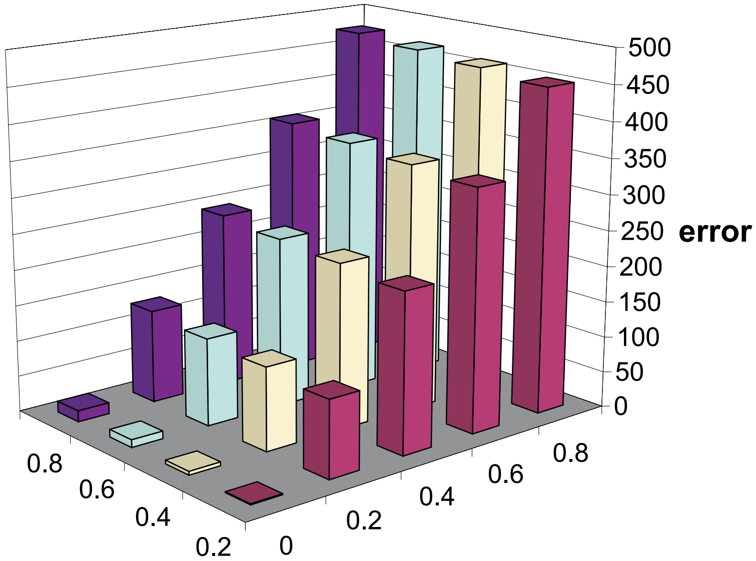

Fig. 23: Maximum error incurred by the decisions returned by the schedulability analysis algorithm, corresponding to the results shown in Figure 22.

3.9 Summary

In this chapter we introduced a task model for network packet processing applications. This model can be used to evaluate the feasibility of a mapping of different packet processing tasks onto the different processing elements of a packet processor. This problem was posed as a schedulability analysis question for the proposed task model. However, it turns out that schedulability analysis for the proposed model is intractable (NP-hard). The main contribution of this work is that we are able to demonstrate that in spite of the intractability of this problem, for all practical purposes it can be solved in polynomial time. Towards this, we have introduced a novel scheme called *approximate schedulability analysis*, which is also applicable to a number of other task models from the real-time systems area, for which the known algorithms for schedulability analysis either have exponential or at best pseudo-polynomial running time.

The problem of designing efficient *false-negative* schedulability tests for the generalized multiframe task model was stated as a possible future research direction in [16]. False-negative tests always identify task sets which are not schedulable, but occasionally return the wrong answer in the case of schedulable task sets. Our work addresses a much more generalized version of this problem. Apart from being of theoretical interest, the experimental results presented in this chapter suggest that the algorithms are implementable and lead to clear benefits in terms of running time.

4

An analytical framework for timing analysis

As mentioned in Chapter 1, a packet processor typically consists of a collection of heterogeneous processing elements onto which the different packet processing tasks are mapped. In Chapter 3 we addressed the feasibility of any such mapping. In this chapter we are concerned with the overall timing behaviour of a packet processor—as packets flow through the system and get processed in the different processing elements, for any given procesing element, what are the timing characteristics of the packet stream flowing out of this element in relation to the input packet stream? How can these input-output relations for the different processing elements be combined in a compositional manner to derive the timing properties of the whole system? Answers to these questions lead to several insights about the system properties. These include the determination of the on-chip buffer-memory requirements and the off-chip memory bandwidth, which can be computed from the maximum number of packets that can be inside the system at any point in time. Other such properties include the utilization of different buses and processors, which can be determined from the maximum and minimum number of packets that cross any of these resources within a specified amount of time.

The main difficulties which any analysis method faces while attempting to answer the above questions stem from the following facts. Packet processors are heterogeneous in nature, where different processing and communication elements have different interfaces and implement different scheduling and resource sharing strategies, thereby making it difficult to design any compositional analysis scheme. Secondly, there is a large degree of parallelism in the system because the different processing elements work concurrently, and as explained in Chapter 2, the input packet stream being processed is composed of several interleaved flows where packets from any flow need to be processed differently and concurrently with packets from the other flows. Modeling this concurrency and the interfering effects of the different flows on any processing or communication resource calls for new task and system models compared to those used in traditional application domains of real-time embedded systems like digital signal processing. Moreover, as packets from different flows traverse from one processing resource to the next, there might be intermediate bursts and packet jams depending on how the different flows are scheduled on a resource, making the computation of input-output timing characteristics of a flow nontrivial and dependent on the other flows in the system. A detailed discussion of the differences between the packet processing domain and traditional application domains for embedded systems (data-dominated applications) can be found in Chapter 2.

Most of the currently available methods and tools for analysing and evaluating such systems rely on simulation and hence suffer from high running times, incomplete coverage, and failure to identify corner cases. To guarantee certain timing properties using analysis schemes that have reasonable running times, it is necessary to develop static formal analysis methods based on abstract system models. In this chapter we take this approach and study an analytical framework for system-level timing analysis for packet processors having a heterogeneous architecture, in which the different packet processing tasks corresponding to the different flows have already been mapped onto the different processing resources. The framework is based on an abstract model of the architecture, a model for packet processing tasks which is essentially similar to the one introduced in Chapter 3, a model for packet flows which is based on the model introduced in Chapter 2, and a calculus for reasoning about such systems. This calculus, which we refer to as *real-time calculus*, was introduced by Thiele *et al.* in [145]. The application of this calculus to analyse packet processing architectures was first shown by Thiele *et al.* in [144], and subsequently more detailed results were presented in [142] and [143]. However, it was not shown how the results from this framework compare with the theoretical results that can be derived using already known techniques from the real-time systems area (i.e. whether this framework generalize previous results in any sense, or whether it gives rise to new results not known from the real-time systems literature). Secondly, it is not clear how closely these results match those which can be obtained using detailed system-level or cycle-accurate simulations. Without a clarification of these two issues, the applicability of this framework to analyse any realistic system is not clear. In this context, we make the following contributions in this chapter.

- We show that the results that can be obtained within this framework generalize many results from the real-time system area which are based on standard event models (like periodic, sporadic, etc.) and scheduling disciplines. Using these known results from the real-time systems literature, the packet arrivals at any resource would typically be modeled by a periodic or sporadic event model which would be an approximation of a real packet trace. The event model used in our framework, on the other hand, is a generalization of any of these standard event models, where any real packet trace can be accurately modeled without resorting to approximating the packet arrival times. Secondly, in contrast to current practice, where different analysis techniques are used for different combinations of event models and scheduling disciplines, we show that our framework provides a unified way of analysing any combination of event model and scheduling policy. Further, it also provides a method for extending the timing analysis results to determine other system properties in a single coherent way and hence serves as a general framework for the performance evaluation of packet processing (hardware/software) architectures.

- Through a realistic case study, we also show that the results obtained from this framework compare well with those obtained by detailed cycle-accurate simulations. Whereas the former takes only fractions of a second to determine the properties of an architecture, simulation based approaches usually run in time in the order of several hours.

- Based on the above finding, we propose a new methodology for the design space exploration of system-on-a-chip (SoC) based network packet processors. Currently, most SoC based embedded system architectures rely on simulation as a means for performance evaluation. The design of such systems usually start with a parameterizable template architecture, where the design space exploration is restricted to identifying the suitable parameters of the architectural components (like cache sizes and associativity, and bus widths). The design space exploration of packet processing architectures however involves a combinatorial aspect (arising from questions like which architectural components should be chosen and how should they be interconnected, how should tasks be mapped, etc.), which increases the design space. To cope with this, we hypothesize that any automated or semi-automated design space exploration in this case should be separated into multiple stages, each having a different level of abstraction. Further, it would be appropriate to use an analytical performance evaluation framework, such as the one studied in this chapter, during the initial stages and resort to simulation only when a relatively small set of potential architectures is identified. We support this hypothesis based on the results of the case study that we present in this chapter.

The rest of this chapter is organized as follows. In the next section we motivate the need for analytical frameworks for timing analysis and performance evaluation in the context of design space exploration of hardware/software architectures of packet processors, which is followed by a review of the existing work in this area in Section 4.2. In Section 4.3 we give a model for characretizing packet flows and processing resources by extending the basic models presented in Section 2.3 of Chapter 2. Based on this, we present in Section 4.4 the analytical framework for timing analysis and show how it can be used for the performance evaluation of packet processors. The results pertaining to a comparison of this framework with the results that can be derived from classical scheduling theory is given in Section 4.5. Finally we compare the results of evaluating a realistic packet processing architecture using this framework, with detailed cycle accurate simulations. Towards this, we first describe the simulation setup in Section 4.6, and Section 4.7 presents the results of the comparison. Finally, in Section 4.8 we propose a new methodology for the design space exploration of packet processing architectures based on the results learnt from the comparative study.

4.1 Analytical frameworks in design space exploration

Because of the new and different requirements of packet processors as outlined in Chapters 1 and 2, the design and analysis of the hardware/software architecture of a packet processor calls for specialized modeling techniques and frameworks which do not fall under the preview of traditional embedded processor design. As a result, recently there has been a number of proposals for performance models and design frameworks specific to network packet processing architectures (see [43, 44, 59, 70, 71, 72, 142, 143, 153, 155]). The goal of these frameworks and models is to aid a designer in understanding the performance tradeoffs involved in a design and come up with an optimal architecture that suits the application scenario at hand.

Realizing these goals in the context of traditional embedded processor design typically involves two issues: a method for performance evaluation or analysis of any specified architecture, and means for covering the design space or design space exploration (through which all the possible architectures need to be identified). An evaluation of all the possible architectures during the exploration lead to the optimal design. In most cases it is possible to formulate a parameterized *architecture template*, where the design space exploration is restricted to finding appropriate values of the parameters such as bus width, cache associativity and cache size. The resulting design space is therefore reasonably small and it is usually feasible to exhaustively evaluate all the possible designs by simulation. When the design space is relatively larger, techniques such as partitioning the architecture into disjoint subsystems and using independent design space explorations for the different subsystems have been used [67, 90, 120]. Even in such cases, the choice of the different architectural components is usually fixed (for example, see [3, 128], where the system always consists of a VLIW processor, a systolic array and a cache subsystem and the design space exploration consists of identifying appropriate parameters for each of these components), and the possible mappings of different tasks onto the processing elements are also either fixed or are very restricted in number. This is unlike the case with packet processing architectures where the number of allocations of the processing elements and the possible mappings of tasks onto these elements can potentially be very large. Design space exploration in this context therefore assumes a different complexity.

Packet processing architectures are very heterogeneous in nature and usually it is not possible to define a parameterizable template architecture. As a result the design space is larger compared to those in the case of typical embedded processor architectures and involves a combinatorial aspect in addition to traversing the parameter spaces of the different components. These processors might also be used in multiple application scenarios (such as core or access networks), might require to support different traffic classes (where some classes might have quality-of-service requirements and others have minimum throughput requirements), and at the same time should be flexible to be able to incorporate new functionality. In order to account for all of these issues in the design phase, we believe that new design methodologies are required. In particular, resorting to exhaustive simulations of all possible designs is no longer a feasible option for automated design space exploration. Hence, using other means of performance evaluation such as appropriate analytical models, which should be fast as well as reasonably accurate, is necessary,

4.1.1 Performance evaluation in the context of design space exploration

It is known that typically the design flow of complex systems-on-a-chip (SoC) architectures starts with an abstract description of the application and some performance requirements, which are then used to drive a system-level design space exploration for identifying a suitable architecture. This involves evaluating many prospective architectures on a system-level and an iteration loop between the exploration and the evaluation steps. Once the main architectural decisions have been made, the resulting architectures are then more accurately evaluated, possibly on the basis of many other criteria which were not considered previously. The design space exploration at this stage, in contrast to the previous, might only involve tuning the parameters of different cores in a core-based SoC design.

Here we argue that in the case of heterogeneous SoC architectures, this separation of the design space exploration into multiple stages is all the more important in order to tackle the large and the different nature of the design space. In particular, we hypothesize that in the context of packet processors, the underlying framework for performance evaluation should vary depending on the stage of the design space exploration—it will be more appropriate to use analytical methods during the initial stages and resort to simulation when a relatively small set of promising alternatives has been identified. None of the known performance evaluation frameworks for network packet processors have been evaluated or positioned from this perspective. From a designer's point of view it would be useful to know if any of the known modeling techniques are more suitable for a particular stage of the architecture design.

In any of the design phases, for a potential architecture at hand, the performance evaluation needs to answer questions such as: Does this architecture meet the required line speeds and maximum allowable delays experienced by packets? What are the limits to the improvement in processor or bus utilization as the number of processor cores is increased? How do the cache/memory organization impact these limits? Will a given hardware assist improve the system performance compared to a software implementation? We believe that the exact nature of these questions, how accurately they need to be answered, and what is the allowable computation/simulation time required to answer them strongly depend on the design phase. For packet processors many of these can be adequately answered with a system-level model, and we show that the analytical framework studied in this chapter is suitable for this purpose. Its analysis/evaluation time is orders of magnitude faster when compared to simulation based frameworks and it is hence appropriate for a system-level design space exploration when the design space is very large.

In support of our hypothesis, we compare the results obtained by this framework with detailed cycle accurate simulations of a realistic architecture. Based on the timing analysis results, we consider three performance metrics: (i) the line speed or the end-to-end throughput that can be supported by the architecture, which is measured using the utilization of its different components (processors, buses, etc.) and thereby also identifying which component acts as the bottleneck, (ii) the end-to-end packet latencies, (iii) the on-chip cache/memory requirement of the architecture. Many important questions that arise in the context of packet processors pertain to these metrics. The usefulness of the results obtained from the analytical framework should be evaluated with respect to their relative accuracy when compared with the simulation results, and the time it takes to compute these results compared to simulation times (under the assumption that there is a high confidence in the simulation results).

One of the major criticisms of the analytical framework we consider here, has been that although it is sufficiently general, it still remains to be seen if it can be applied to analyse any realistic network packet processor architecture (see, for example, [151]). Our work in this chapter addresses this issue and additionally places this framework in an appropriate stage of a design flow.

4.2 Existing approaches

There is a large body of work devoted to system-level performance analysis of SoC architectures (see [61] and the references therein). For the evaluation of more specific aspects of an architecture, such as the on-chip communication infrastructure, see [96] and the references therein. However, in this section we focus on the work done specifically in the context of packet processors.

As mentioned in the last section, lately there has been a number of proposals for performance analysis techniques specifically targetted towards architectures for packet processing. These can be broadly classified into (i) purely simulation-based approaches, which include both system-level simulations and cycle-accurate simulations, (ii) trace-based performance analysis, (iii) analysis frameworks using “static analytical models”, and (iv) frameworks based on “dynamic analytical models”.

Purely simulation-based approaches evaluate an architecture by executing one or more packet processing applications on some executable model of the architecture and then use some defined set of stimuli (usually in the form of packet traces) to trigger or drive the applications. Since the chosen set of stimuli can activate only a fixed set of “execution paths”, both in the architecture as well as in the application, this set must be so chosen that it represents some typical scenario (rather than a special case) in the execution. This turns out to be one of the main drawbacks of simulation-based evaluation or analysis. The evaluation results obtained using any given set of input traces are often difficult to interpret in a more broad setting. Further, as we already pointed out, in the context of design space exploration, obtaining meaningful simulation results usually call for detailed models which involve complex simulation setups and high simulation times. When the underlying architecture design space is relatively large, such detailed and time consuming simulations are therefore not feasible. Lastly, another major problem with simulations is the need for an executable model. For many architectural components such models may not be available.

However, the main advantage of using simulation as a means for performance evaluation is that many dynamic and complex interactions in an architecture can be taken into account, which are otherwise difficult to model analytically. As a result, if the executable model being simulated is sufficiently detailed, then there can be a high confidence in the results obtained.

A performance evaluation scheme using system-level simulations of the architecture of a packet processor is described in [70]. This work focusses on packet processors to be used in access networks where a customer’s traffic enters the network of a service provider. The simulations take into account the different packet processing tasks, the memory architecture, the different architectural components or building blocks such as processors, hardware assists, etc., and the dynamic behavior of the different packet flows as packets traverse from one resource to the other. Therefore, the backlog in the queues in front of different resources resulting from packets waiting to get processed, the burstiness of packet flows inside the processor, how fairly the different flows are served by the different resources, and the effects of different scheduling and arbitrations schemes on these can be measured.

The modeling framework presented in [43] is based on cycle accurate models of the different programmable processing elements that constitute a packet processor. The approach is to approximate the packet processing application characteristics statistically, based on simulation and instruction profiling and then use those approximations to form system resource usage and contention estimates.

The framework is composed of independent application, system and traffic models. The application is modeled using the Click modular router from MIT [94]. Click consists of a collection of software modules for describing various router functionality. Such modules in Click are called *elements*, and by putting together different elements in the form of a graph (which is called a *configuration*) it is possible to construct applications such as IP routers, firewalls, QoS routers, etc. The architecture to be evaluated is simulated using SimpleScalar [25] and it implements the Alpha instruction set [6]. Click modules are compiled for the Alpha ISA and the compiled modules are then executed on this architecture. By simulating this execution using different traffic traces, the profiled code yields various information such as instruction count, details regarding cache behavior, etc. These are then used to compute various performance metrics for the architecture being evaluated, related to packet latency, bandwidth and resource utilization. It may be noted that results such as execution times, obtained directly from the simulation, form only a part of the overall packet processing/forwarding times. These must later be adjusted to account for contention for the shared resources and the costs of synchronization overheads since they can not be modeled in the SimpleScalar toolkit. Moreover, for elements which do not have any executable model or software implementation (such as dedicated hardware units) and hence can not be simulated, the profile and external dependencies need to be provided manually by the user.

In contrast to this approach, the work done in [155] models an architecture in SystemC [73]. This work mostly focuses on the communication subsystem and the memory organization of an architecture. The models are then simulated on packet traces and performance metrics such as bus utilization, memory fill levels, and packet delays are evaluated. This work forms the basis of the simulations results that we present in this chapter and further details on it are given in Section 4.6.

These two approaches as they exist now are complementary to each other. [43] uses an accurate processor model but a very elementary model of the communication subsystem of an architecture (such as buses, bridges, etc.). On the other hand, the framework in [155] implements cycle-accurate models for buses and bridges, but has a very simple processor and application model.

Clearly, there exists an opportunity to combine the above frameworks which will then consist of a detailed model of processors as well as other components of the communication subsystem, such as buses, for which SystemC models already exist. In the same way as there exists a SimpleScalar model of the Alpha processor, there already exists a SystemC model of PowerPC which can be simulated on executable code. This opens the possibility of integrating this model with the SystemC models of the communication subsystem used in [155], and then use detailed application models by compiling Click modules for the PowerPC instruction set as is done in [43].

In order to reduce the simulation times, many approaches gather execution characteristics of applications which do not change from one one architecture to another, based on a single initial simulation. These are then used to evaluate different architectures without resorting to further detailed simulations. In particular, such *trace-based approaches* simulate the execution of a program on an abstract model of an architecture and collect statistics such as the number of memory and bus accesses. These statistics are then used to evaluate the performance of any concrete architecture. The performance of this class of approaches, both in terms of the running time involved and the quality of the results obtained, is usually intermediate between detailed simulations and approaches based on anytical models which perform a static analysis of the system.

An example of this method in the context of evaluating the communication architecture of a SoC design can be found in [96]. Here an initial cosimulation of the system is performed with the communication described in an abstract manner (as events or abstract data transfers), and a set of traces is extracted from the simulation which contain the details of the communication between the different architectural components. This information is represented in the form of a "communication analysis graph" (CAG) which is manipulated based on the concrete architecture (its topology, the mapping of different tasks onto the processors, and the arbitration scheme used in each communiation channel). The result of manipulating the CAG is an estimate of the system performance and provides various statistics about the communication in the concrete architecture. As in the case of any usual system simulation, the derived results are specific to the input stimuli used, and hence this must be carefully chosen.

In the context of packet processors, the work described in [59], in some sense, can be considered to fall into this category of trace-based approaches. The performance analysis requires an initial characterization of a set of benchmarks (related to packet processing applications and workloads) using detailed exhaustive simulation runs for a range architecture organizations (such as organizations of the cache/memory subsystem). The information extracted from these runs such as instruction and data miss rates, load/store instruction shares, etc. is then fed into analytical models of the different architectures being evaluated, to calculate the utilization of different resources, layout area requirements of the architectures, and the performance of the architectures. This framework has been extended in [60] to also reason about the power requirements of an architecture.

In the architecture model considered in [59], the different components that make up the architecture and the interconnection among these components (the so called *architecture template*) is fixed. The architecture template consist of a number of multithreaded processors organized in clusters. Each cluster consists of a number of processors, each having its own cache, and the cluster communicates with an off-chip memory using its own memory interface. The parameters that can be changed while deriving different concrete architectures from the architecture template are the number of threads running in each processor, the cache sizes, the number of processors in each cluster, the number of clusters in the network processor, the width of the memory channels, etc. For evaluating an architecture with a given set of parameters, an analytical model for multithreaded processors which was proposed in [4] is used. The benchmark workload and applications which are used to derive the inputs to the analytical model are based on the CommBench benchmark suite [152] and consists of a mix of header-processing and payload-processing applications. Since the architecture template which forms the basis of this framework is fixed, the main focus of this work can be considered to be the cache/memory subsystem of a packet processing architecture.

The underlying analytical model used in [59] (which is based on the model in [4]) can be considered to be a "static analytical model". Such models describe the computation, communication and memory resources using purely algebraic equations whose parameters are static data transfer rates between different architectutal components and also static processing capacities of the different processing elements. Typically these models do not take into account the variations of the processing capacities of a resource over time (possibly due to interrupts or bursts of packets arriving from a flow) and the dynamic interactions between different packet flows on a shared resource. Therefore, although they are simple and provide a fast estimation, the generated results are either overly pessimistic or provide only a poor approximation of the performance of the real system and can differ widely from simulation results. In contrast to these approaches, the model we study in this chapter may be termed as a "dynamic analytical model" since it extends static models to account for the interactions between the different packet flows on a shared resource and the resulting nondeterminism in processing packets from any given flow. Such dynamic analytical models may either be based on statistical methods and queuing theory which typically does an average-case analysis, or they may be based on a deterministic worst-case analysis using methods such as *network calculus* [22] or *real-time calculus* [144, 145]. The framework studied in this chapter follows the later approach of a worst-case analysis. It uses analytical models for both, the hardware/software architecture of a packet processor which takes into account the different scheduling and arbitration policies at the different processing and communication resources, and the different packet flows that are processed by the architecture. Moreover, in contrast to the work in [59], the architecture template or topology is not assumed to be fixed, and therefore different combinations of processors and buses and interconnections between them can be evaluated. The details of this framework are presented in the next two sections.

4.3 Modeling packet flows and resource capacities

Recall from Section 2.3 of Chapter 2 the model for specifying bounds on packet arrivals from any flow using the concept of *arrival curves*. In this section we extend this model to characterize packet flows entering a packet processor. Based on similar concepts, we also introduce a model for describing the computation/communication capacities of the different architectural elements that are used to process/transmit the packets entering these elements.

Let f be a flow entering a given resource. The resource is either a computation resource such as a processor on which some packet processing task is implemented, or it is a communication resource such as a bus that is used to transmit packets between two computation resources or a computation resource and a memory module. Let $R_f(t)$ denote the number of packets from f arriving at the resource during the time interval $[0, t]$. The maximum number of packets arriving at the resource is assumed to be bounded by a right-continuous subadditive function called the *upper arrival curve* denoted by α_f^u. Similarly, a lower bound on the number of packets arriving at the resource is given by a *lower arrival curve* denoted by α_f^l. α_f^l and α_f^u together can be referred to as the *traffic constraint functions* and they satisfy the following inequality.

$$\alpha_f^l(t-s) \leq R(t) - R(s) \leq \alpha_f^u(t-s), \qquad \forall 0 \leq s \leq t$$

For any $\Delta \geq 0$, $\alpha_f^l(\Delta) \geq 0$ and $\alpha_f^l(0) = \alpha_f^u(0) = 0$. Therefore, $\alpha_f^l(\Delta)$ gives a lower bound on the number of packets that can arrive at the resource from the flow f within any time interval of length Δ. $\alpha_f^u(\Delta)$ gives the corresponding upper bound.

Similar to the arrival curves describing packet flows, the computation or communication capability of a resource is described using *service curves*. Given a resource r, let $C_r(t)$ denote the number of packets (or the number of bytes, depending on the resource) that can be processed by r during the time interval $[0, t]$. Then the upper and lower service curves β_r^u and β_r^l describing the processing capabilities of the resource satisfy the following inequality.

$$\beta_r^l(t-s) \leq C_r(t) - C_r(s) \leq \beta_r^u(t-s), \qquad \forall 0 \leq s \leq t$$

Further, for any $\Delta \geq 0$, $\beta_r^l(\Delta) \geq 0$ and $\beta_r^l(0) = \beta_r^u(0) = 0$. Therefore, $\beta_r^l(\Delta)$ is a lower bound on the number of packets that can be processed by r (or the number of bytes that can be transmitted in case r is a communication resource) within any time interval of length Δ. Likewise, $\beta_r^u(\Delta)$ is the corresponding upper bound. Hence, the processing capability of r over any time interval of length Δ is always greater then or equal to $\beta_r^l(\Delta)$ and less than or equal to $\beta_r^u(\Delta)$.

As mentioned above, the arrival curves of a flow f entering a resource r is described in terms of either the number of bytes entering r within any time interval of length Δ or the number of packets within any time interval of length Δ. This depends on the packet processing task implemented on r. For a task such as packet header processing, where the load on r depends on the number of packets entering and not on the sizes of the packets, the arrival curves are defined in terms of number of packets. On the other hand, if r is a communication resource such as a bus, or a payload processing task such as encryption is implemented on r, then the arrival curves are defined in terms of number of bytes.

Now suppose that for each packet (or each byte, as the case maybe) entering the resource r, w units of processing resource have to be spent by r to process the packet. This might be described as the number of processor cycles, bus cycles, or processor instructions. Then if a flow f has arrival curves $\bar{\alpha}_f^l$ and $\bar{\alpha}_f^u$

described in terms of number of packets, these may be transformed as follows to represent the processing request demanded by f from the resource r.

$$\alpha_f^l = w\bar{\alpha}_f^l, \qquad \alpha_f^u = w\bar{\alpha}_f^u$$

Hence, α_f^l and α_f^u now describe the arrival curves of flow f in terms of the processing request (for example, number of processor cycles) demanded from r. If β_r^l and β_r^u describe the processing capability of r in terms of the same units (i.e. processor cycles) then the maximum delay and maximum backlog suffered by packets of flow f at the resource r can be given by the following inequalities.

$$\begin{aligned} delay &\leq \sup_{t\geq 0}\left\{\inf\{\tau \geq 0 : \alpha_f^u(t) \leq \beta_r^l(t+\tau)\}\right\} && (4.1)\\ backlog &\leq \sup_{t\geq 0}\{\alpha_f^u(t) - \beta_r^l(t)\} && (4.2)\end{aligned}$$

A physical interpretation of these inequalities can be given as follows: the delay experienced by packets waiting to be served by r can be bounded by the maximum horizontal distance between the curves α_f^u and β_r^l, and the backlog is bounded by the maximum vertical distance between them.

The packets of flow f, after being processed by the resource r, emerge out of this resource. Let $\alpha_f^{u\prime}$ and $\alpha_f^{l\prime}$ be the upper and lower arrival curves of this processed flow. Similarly, let $\beta_r^{u\prime}$ and $\beta_r^{l\prime}$ be the upper and lower remaining service curves of the resource r, denoting the remaining processing capability of r after processing packets of flow f. Then these can be given as follows (see [144] for further details and proofs).

$$\begin{aligned} \alpha_f^{l\prime}(\Delta) &= \inf_{0\leq t\leq\Delta}\left\{\alpha_f^l(t) + \beta_r^l(\Delta - t)\right\} && (4.3)\\ \alpha_f^{u\prime}(\Delta) &= \inf_{0\leq t\leq\Delta}\{\sup_{v\geq 0}\left\{\alpha_f^u(t+v) - \beta_r^l(v)\right\} \\ &\qquad\qquad +\beta_r^u(\Delta - t), \beta_r^u(\Delta)\} && (4.4)\\ \beta_r^{l\prime}(\Delta) &= \sup_{0\leq t\leq\Delta}\left\{\beta_r^l(t) - \alpha_f^u(t)\right\} && (4.5)\\ \beta_r^{u\prime}(\Delta) &= \sup_{0\leq t\leq\Delta}\left\{\beta_r^u(t) - \alpha_f^l(t)\right\} && (4.6)\end{aligned}$$

The maximum utilization of the resource r due to the processing of flow f is given by:

$$utilization \leq \lim_{\Delta\to\infty}\frac{\beta_r^u(\Delta) - \beta_r^{l\prime}(\Delta)}{\beta_r^u(\Delta)} \qquad (4.7)$$

4.4 A model for timing analysis and performance evaluation

The modeling concepts used in the last section are known from work done in the area of communication networks. The results derived on the basis of these models (see also [144]) have their background in network calculus [22] which use the concept of arrival and service curves in a network theoretic context. In this section we show how these results can be used to formulate an analytical performance evaluation model for network processors. The results here were originally derived in [142] and [143] and we refer the reader to these papers for additional details.

Here we view a network processor as a collection of different processing elements (such as CPU cores, micro-engines, and dedicated units like hardware classifier, cipher, etc.) and memory modules connected to each other by communication buses. On each of these processing elements one or more packet processing tasks are implemented. Depending on the sequence in which these tasks process a packet, and the mapping of these tasks on to the different processing elements of the network processor, any packet entering the network processor follows a specific sequence through the different processing elements. The flow to which this packet belongs is associated with its arrival curves. Similarly all the resources have their associated service curves. As the packets of the flow move from one processing element to the next, and also cross communication resources such as buses, both, the arrival curves of the flow and the service curves of the resources get modified following the Equations (4.3)-(4.6) given in Section 4.3. Given this, the maximum end-to-end delay experienced by any packet, the on-chip memory requirement of the network processor, and the utilization of the different resources (both computation and communication) can now be computed using Equations (4.1), (4.2) and (4.7).

To formally state the above procedure, consider that for the set of flows F entering the network processor, there is a task graph $G = (V, E)$. Any vertex $v \in V$ denotes a packet processing (or communication) task. For any flow $f \in F$, let $V(f) \subseteq V$ denote the set of packet processing tasks that have to be implemented on any packet from F. Additionally, a subset of directed edges from E defines the order in which the tasks in $V(f)$ should be implemented on any packet from f. Therefore, if $u, v \in V(f)$ represent two tasks such that for any packet belonging to f, the task v should be implemented immediately after task u, then the directed edge (u, v) belongs to the set E. Hence, for each flow f there is a unique path through the graph G starting from one of its source vertices and ending at one of its sink vertices. The vertices on this path represent the packet processing tasks that are to be implemented on packets from f.

Figure 24 shows a hypothetical network processor architecture built out of blocks from an existing core library [80, 81]. Here PPC refers to the PowerPC 440 core, and PLB and OPB refer to two buses called the Processor Local Bus and the On-chip Peripheral Bus provided by the CoreConnect [81] architecture for interconnecting cores and custom logic. The numbers on the arrows in this figure either indicate actions that are to be performed by the different blocks as a packet flows through the architecture, and they are ordered according to the numbering.

From Figure 24 it is possible to construct a task graph considering the appropriate packet transfers from one resource to another. This task graph can then

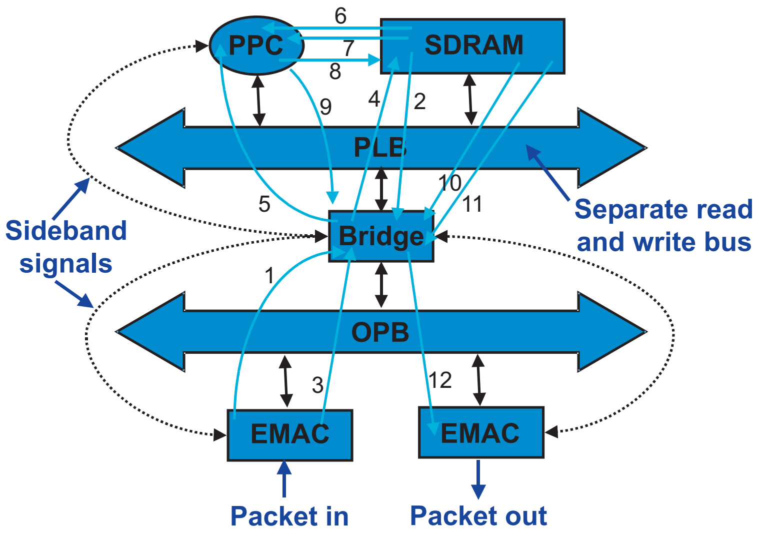

Fig. 24: A system-level model of a network processor. The figure shows the path that a packet follows through the architecture. The numbers on the arrows indicate the different actions involved (which are explained in Table 3) while the packet travels through the architecture, and specify the order in which these actions are executed. Further details of this model can be found in Section 4.7.

be used to compute the load on the different buses (such as the OPB and the PLB), the on-chip memory requirement of this architecture to store the buffered packets in front of each resource, and the end-to-end packet delays.

4.4.1 Analysis using a scheduling network

In Section 4.3 we described how to compute the delay and backlog experienced by a flow passing through a single resource node processing the flow. For this we characterized the flow using its arrival curves and the resource node using its service curve and also derived formulas for the maximum utilization of this resource and the outgoing arrival and resource curves. Now we extend these results to consider the case where the flow passes through multiple resource nodes as shown in Figure 24.

The outgoing arrival curve capture the characteristics of the processed packet flow (for example its burstiness and long term rate), which might be different from the characteristics the flow has before entering the resource. Similarly the outgoing service curve indicates the remaining process capability of the resource after processing the flow. The idea now is to use this outgoing arrival curve as an input to another resource node (more precisely, the resource node where the next packet processing task as given by task graph described above is implemented). In the same way, the outgoing service curve of the first resource is used to process packets from a possibly second flow. This procedure can be illustrated using a *scheduling network*. For example, Figure 25 shows the scheduling network corresponding to the packet traversal through the archi-

Step	Action
1	Sideband signal from EMAC to bridge (indicating that a new packet has arrived)
2	Bridge gets a ”buffer descriptor”(BD) from the SDRAM
3	Packet is sent from EMAC to bridge over the OPB
4	Packet is sent from bridge to SDRAM over the PLB write bus
5	Sideband signal from Bridge to PPC (indicating that the new packet has been stored)
6	CPU get buffer descriptor over the PLB read bus
7	CPU gets packet header over the PLB read bus
8	CPU processes header, and stores it back to SDRAM over the PLB write bus
9	Sideband signal from bridge to CPU (indicating that the packet can be sent out)
10	Bridge gets buffer descriptor over the PLB read bus
11	Bridge gets packet over the PLB read bus
12	Packet sent out to specified a EMAC over the OPB

Tab. 3: Sequence of actions for every processed packet in the architecture model shown in Figure 24.

tecture shown in Figure 24.

In general, multiple flows enter a network processor and are processed by the different resources in the order specified by the task graph described above. As packets from several flows arrive at a resource, they are served in an order determined by the scheduling policy implemented at the resource. For example, many buses use a fixed priority bus arbitration scheme. Other scheduling policies might be FCFS and round robin. We illustrate the analytical model here assuming that all the resources use fixed priority. However, the model can be extended to incorporate other scheduling policies as well.

Let us assume that there are n flows $f_1, \ldots, f_n$ arriving at a resource r, which serves these flows in the order of decreasing priorities, i.e. f_1 has the highest priority and f_n the lowest. For each packet of the flow f_i, some packet processing task t_i implemented on r processes the packet and this requires $w(t_i, r)$ processing units from r. For example, $w(t_i, r)$ might be the number of processor instructions, or bus cycles in case r is a communication resource. We henceforth denote $w(t_i, r)$ by w_i when it is clear which resource is being referred to. Each flow f_i arriving at r is associated with its upper and lower arrival curves $\bar{\alpha}_i^u$ and $\bar{\alpha}_i^l$ respectively and receives a *service* from r which can be bounded by the upper and lower service curves β_i^u and β_i^l respectively. The service available from r in the unloaded state (i.e. before any of the flows $f_1, \ldots, f_n$ are served) is bounded by the upper and lower service curves β^u and β^l respectively.

In the fixed priority scheme r services the flows in the order of decreasing priorities and the remaining service curve after processing a flow is used to

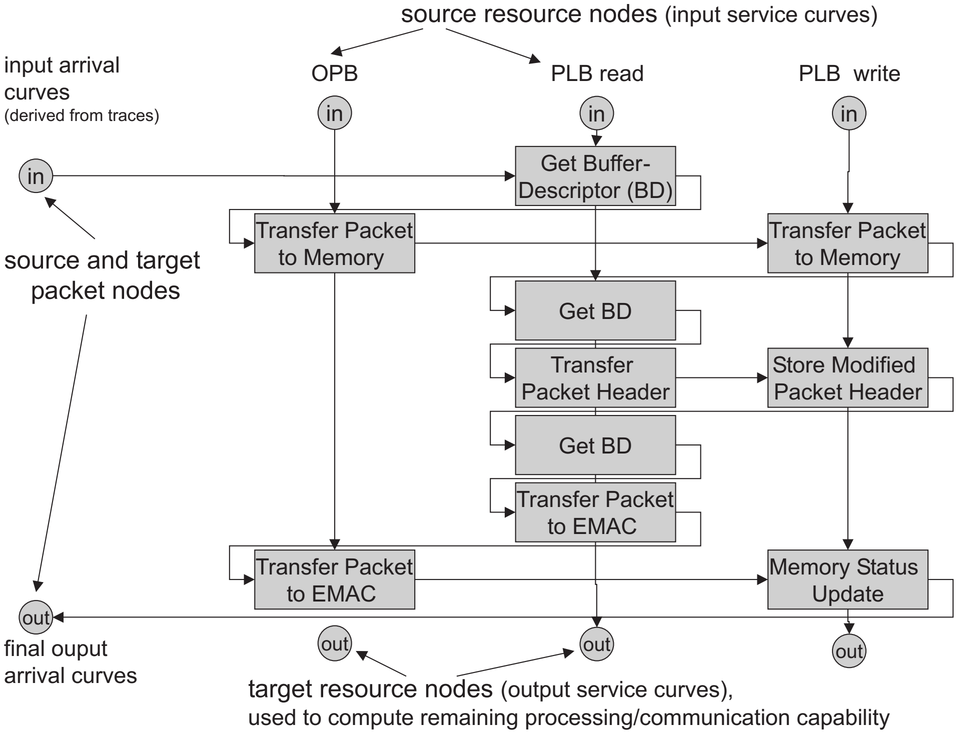
source resource nodes (input service curves)
input arrival curves
(derived from traces)
OPB
PLB read
PLB write
in
Get Buffer-Descriptor (BD)
Transfer Packet to Memory
source and target packet nodes
Get BD
Transfer Packet Header
Store Modified Packet Header
Transfer Packet to EMAC
Memory Status Update
out
final ouput arrival curves
target resource nodes (output service curves), used to compute remaining processing/communication capability

Fig. 25: The scheduling network for the architecture given in Figure 24.

serve the lower priority flows. The resulting arrival curves and the remaining service curves can be computed using Equations (4.3)-(4.6) given in Section 4.3. Since packets from different flows might have different processing requirements given by $w_1, \ldots, w_n$, the arrival curves first need to be scaled as described in Section 4.3. Similarly, the outgoing arrival curves need to be scaled back as follows. If $\alpha_i^{u\prime}$ and $\alpha_i^{l\prime}$ are the outgoing arrival curves from a resource node calculated using Equations (4.3) and (4.4), then $\overline{\alpha}_i^{u\prime} = \lceil \alpha_i^{u\prime}/w_i \rceil$ and $\overline{\alpha}_i^{l\prime} = \lfloor \alpha_i^{l\prime}/w_i \rfloor$. The floor/ceiling functions are used since a subsequent resource node can start processing a packet only after the task implemented on r finishes processing it.

Finally, given the service curves for the unloaded resource β^u and β^l, and the arrival curves $\bar{\alpha}_i^u$, $\bar{\alpha}_i^l$, $i = 1, \ldots, n$, we show how the service curves β_i^u and β_i^l for $i = 1, \ldots, n$ can be determined. As described before,

$$\begin{aligned} \alpha_i^u &= w_i \bar{\alpha}_i^u, \qquad \alpha_i^l = w_i \bar{\alpha}_i^l, \qquad i = 1, \ldots, n \\ \overline{\alpha}_i^{u\prime} &= \lceil \alpha_i^{u\prime}/w_i \rceil, \qquad \overline{\alpha}_i^{l\prime} = \lfloor \alpha_i^{l\prime}/w_i \, \rfloor, \qquad i = 1, \ldots, n \end{aligned}$$

$$\begin{aligned} \beta_1^u &= \beta^u, \qquad \beta_1^l = \beta^l \\ \beta_i^u &= {\beta_{i-1}^u}', \qquad \beta_i^l = {\beta_{i-1}^l}', \qquad i = 2, \ldots, n \end{aligned}$$

where ${\beta_{i-1}^u}'$ and ${\beta_{i-1}^l}'$ for $i = 2, \ldots, n$ are determined from β_{i-1}^u, β_{i-1}^l, α_{i-1}^u and α_{i-1}^l by applying Equations (4.5) and (4.6) from Section 4.3. Lastly, the remaining service curve after processing all the flows is given as follows.

$$\beta^{u\prime} = \beta_n^{u\prime}, \qquad \beta^{l\prime} = \beta_n^{l\,\prime}$$

These can be used to compute the maximum utilization of the resource using the inequality (4.7). The processed flows with their resulting arrival curves $\overline{\alpha}_i^{u\prime}$ and $\overline{\alpha}_i^{l\,\prime}$ now enter other resource nodes for further processing.

4.4.2 Scheduling network construction

Using the results in the last section we now describe the procedure for constructing a scheduling network. This can then be used to determine properties of the architecture such as the on-chip memory requirement, the end-to-end delay experienced by packets and the utilization of the different on-chip resources such as processors and buses.

The inputs necessary for constructing such a network are the task graph which denotes for each flow the sequence of packet processing tasks that are to be executed on any packet of the flow and the target architecture on which these tasks are mapped.

The scheduling network contains one *source resource node* and one *target resource node* for each resource used in the architecture. Similarly, for each packet flow there is a *source packet node* and a *target packet node*. For each packet processing task of a flow there is a node in the network marked with the task and the resource on which this task is implemented. For two consecutive tasks u and v of a flow, if u is implemented on a resource r_u and v on a resource r_v then there is an edge (drawn horizontally in the Figure 25) in the scheduling network from the node (u, r_u) to (v, r_v). For a given flow, if u and v are two tasks implemented on the same resource r and u precedes v in the task graph, then there is an edge (drawn vertically in the Figure 25) from the node (u, r) to the node (v, r).

The arrival curves of the flows and the service curves of the resources pass from one node to the next in the scheduling network and get modified in the process, following Equations (4.3)-(4.6).

For a given flow f, let α_f^u be its upper arrival curve before entering the network processor. Suppose this flow passes through nodes of the scheduling network which have their input lower service curves equal to $\beta_1^l, \ldots, \beta_m^l$. Then the *accumulated lower service curve* β^l used to serve this flow can be computed as follows.

$$\begin{aligned}
\bar{\beta}_1^l &= \beta_1^l \\
\bar{\beta}_{i+1}^l &= \inf_{0 \le t \le \Delta} \left\{ \bar{\beta}_i^l(t) + \beta_{i+1}^l(\Delta - t) \right\}, \; i = 2, \ldots, m-1 \\
\beta^l &= \bar{\beta}_m^l
\end{aligned}$$

Now the maximum end-to-end delay and the total backlog experienced by

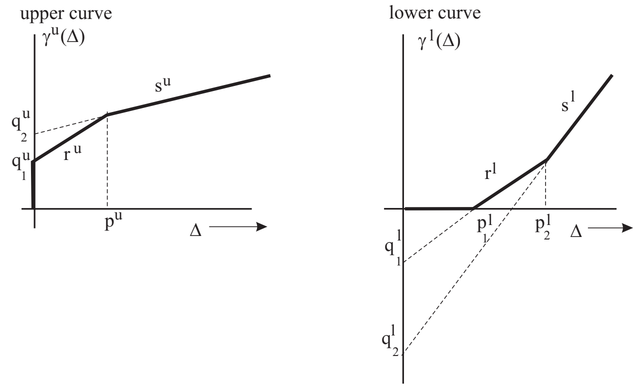

Fig. 26: Piecewise linear approximations of the upper and lower (arrival and service) curves.

packets from the flow f can be given by:

$$delay \leq \sup_{t \geq 0} \left\{ \inf\{\tau \geq 0 : \alpha^u_f(t) \leq \beta^l(t+\tau)\} \right\} \tag{4.8}$$

$$backlog \leq \sup_{t \geq 0} \{\alpha^u_f(t) - \beta^l(t)\} \tag{4.9}$$

Compared independently deriving the delay and backlog at single resources using inequalities (4.1) and (4.2) from Section 4.3 and adding them, the inequalities (4.8) and (4.9) give tighter bounds.

4.4.3 Approximating the arrival and service curves

The Equations (4.3)-(4.6) are clearly expensive to compute for general arrival and service curves. Moreover, these equations need to be computed for all the nodes of a scheduling network. Additionally, if these curves are to be meaningfully derived out of packet traces (as shown later in this chapter), then the resulting curves can be described by a few parameters such as the maximum packet size, the short-term burst rate, and the long-term packet arrival rate. In view of this, we propose a piecewise linear approximation of all arrival and service curves. Using these approximations, the Equations (4.3-(4.6) can be efficiently computed, thereby avoiding the computational bottleneck involved in dealing with general curves.

Each curve in this case is approximated using a combination of three straight line segments. This allows us to exactly model an arrival curve in the form of a T-SPEC [125], which is widely used in the area of communication networks. Figure 26 shows the resulting form of the upper and lower curves (both arrival and service). Here q^u_1 represents the maximum possible load on a resource for processing one packet. The slope r^u of the middle segment of the upper curve can be interpreted as the burst rate, and the slope s^u as the (load on a resource due to the) long term packet arrival rate. In the case of communication resources, q^u_1 represents the maximum packet size. The values of p^u and p^l_1, p^l_2 can be computed from these parameters.

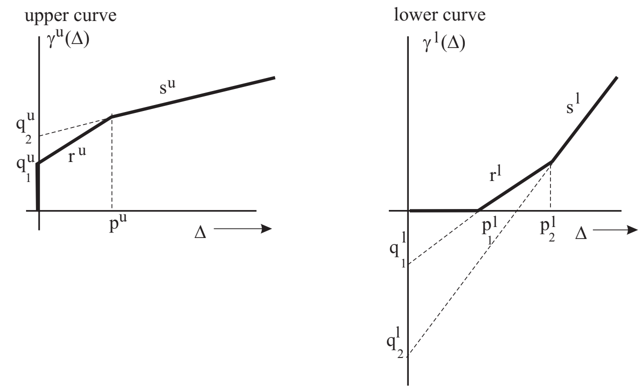

Fig. 27: Improved approximation of upper and lower curves.

Any upper (say γ^u) and lower (say γ^l) curves can now be written as the following.

$$\begin{aligned}\gamma^u(\Delta) &= \min\{q_1^u + r^u\Delta, q_2^u + s^u\Delta\}\\ \gamma^l(\Delta) &= \max\{q_2^l + s^l\Delta, q_1^l + r^l\Delta, 0\}\end{aligned}$$

where,

$$\begin{array}{lll} q_2^u \geq q_1^u \geq 0, & r^u \geq s^u \geq 0, & r^u = s^u \Leftrightarrow q_1^u = q_2^u \\ q_2^l \leq q_1^l \leq 0, & 0 \leq r^l \leq s^l, & r^l = s^l \Leftrightarrow q_1^l = q_2^l \end{array}$$

Using these curves, the Equations (4.3)-(4.6) as well as maximum delay and backlog can be evaluated efficiently using symbolic techniques.

4.4.4 Improved approximations

In this section we show that it is possible to obtain improved approximations of the remaining arrival and service curves, by approximating these curves using three line segments instead of two as in Section 4.4.3. The resulting calculations however become more involved in this case. Figure 27 shows the resulting arrival and service curves. This allows us to exactly model an arrival curve in the form of a T-SPEC [125]. In the case of an arrival curve, q_1^u may represent the maximum possible workload involved in processing a single packet, r^u can be interpreted as the burst rate and s^u the long term arrival rate. In the case of communication resources, q_1^u represents the maximum size of a packet.

The upper and the lower curves in this case can be written as:

$$\begin{aligned}\gamma^u(\Delta) &= \min\{q_1^u + r^u\Delta, q_2^u + s^u\Delta\}\\ \gamma^l(\Delta) &= \max\{q_2^l + s^l\Delta, q_1^l + r^l\Delta, 0\}\end{aligned}$$

where,

$$\begin{array}{lll} q_2^u \geq q_1^u \geq 0, & r^u \geq s^u \geq 0, & r^u = s^u \Leftrightarrow q_1^u = q_2^u \\ q_2^l \leq q_1^l \leq 0, & 0 \leq r^l \leq s^l, & r^l = s^l \Leftrightarrow q_1^l = q_2^l \end{array}$$

The values of p^u and p_1^l, p_2^l (see Fig. 27) can be calculated as:

$$p^u = \begin{cases} \frac{q_2^u - q_1^u}{r^u - s^u} & \text{if } r^u > s^u \\ 0 & \text{if } r^u = s^u \end{cases}$$

$$p_1^l = \begin{cases} -\frac{q_1^u}{r^l} & \text{if } r^l > 0 \\ 0 & \text{if } r^l = 0 \end{cases}, \qquad p_2^l = \begin{cases} \frac{q_2^l - q_1^l}{r^l - s^l} & \text{if } r^l < s^l \\ p_1^l & \text{if } r^l = s^l \end{cases}$$

We denote the curves γ^u and γ^l in this case by $U(q_1, q_2, r, s)$ and $L(q_1, q_2, r, s)$ respectively.

Thm. 14: *Given the upper arrival and lower service curves $\alpha^u = U(q_{1\alpha}, q_{2\alpha}, r_\alpha, s_\alpha)$ and $\beta^l = L(q_{1\beta}, q_{2\beta}, r_\beta, s_\beta)$ respectively, the approximate remaining service curve $\beta^{l'} = L(q_1, q_2, r, s)$ can be given by the following four cases.*

1: There exists a $\Delta' > 0$, such that $q_{2\beta} + s_\beta \Delta' = q_{2\alpha} + s_\alpha \Delta'$, and for all $\Delta < \Delta'$, $\alpha^u(\Delta) > \beta^l(\Delta)$. In this case, $r = 0, s = s_\beta - s_\alpha$ and $q_1 = 0, q_2 = q_{2\beta} - q_{2\alpha}$

2: There exists a $\Delta' > 0$, such that $q_{1\beta} + r_\beta \Delta' = q_{2\alpha} + s_\alpha \Delta'$, and for all $\Delta < \Delta'$, $\alpha^u(\Delta) > \beta^l(\Delta)$. In this case, $r = r_\beta - s_\alpha, s = s_\beta - s_\alpha$ and $q_1 = q_{1\beta} - q_{2\alpha}, q_2 = q_{2\beta} - q_{2\alpha}$

3: There exists a $\Delta' > 0$, such that $q_{1\beta} + r_\beta \Delta' = q_{1\alpha} + r_\alpha \Delta'$, and for all $\Delta < \Delta'$, $\alpha^u(\Delta) > \beta^l(\Delta)$. In this case, $r = r_\beta - r_\alpha, s = s_\beta - s_\alpha$ and $q_1 = q_{1\beta} - q_{1\alpha}, q_2 = q_{2\beta} - q_{2\alpha}$

4: There exists a $\Delta' > 0$, such that $q_{2\beta} + s_\beta \Delta' = q_{1\alpha} + r_\alpha \Delta'$, and for all $\Delta < \Delta'$, $\alpha^u(\Delta) > \beta^l(\Delta)$. In this case, $r = s_\beta - r_\alpha, s = s_\beta - s_\alpha$ and $q_1 = q_{2\beta} - q_{1\alpha}, q_2 = q_{2\beta} - q_{2\alpha}$

If $\alpha^u(\Delta) \geq \beta^l(\Delta)$ for all $\Delta \geq 0$ then $r = s = 0$ and $q_1 = q_2 = 0$

Proof: To prove that $\beta^{l'} = L(q_1, q_2, r, s)$ is a valid lower remaining service curve, we shall as before show that $L(q_1, q_2, r, s)(\Delta) \leq \sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\}$ for all $\Delta \geq 0$.

Firstly, it may be noted that $\beta^l(\Delta)$ and $\alpha^u(\Delta)$ are convex and concave respectively. Therefore, $\beta^l(\Delta) - \alpha^u(\Delta)$ and $\sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\}$ are convex. However, in contrast to our approximations with two segments in Section 4.4.3, here we have to consider four different cases.

Case 1 is when the last segment of $\beta^l(\Delta)$ intersects the last segment of $\alpha^u(\Delta)$, at say $\Delta = \Delta'$ (see Figure 28(a)). Therefore, for all $\Delta < \Delta'$, $\beta^l(\Delta) < \alpha^u(\Delta)$. Hence, $\sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\} \leq 0$ for all $\Delta \leq \Delta'$, and therefore $r = 0$ and $q_1 = 0$. When $\Delta \to \infty$, $\sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\}$ $= \sup_{0 \leq u \leq \Delta}\{q_{2\beta} + s_\beta u - q_{2\alpha} - s_\alpha u\}$. Therefore, we have $s = s_\beta - s_\alpha$ and $q_2 = q_{2\beta} - q_{2\alpha}$.

Case 2 is when the middle segment of $\beta^l(\Delta)$ intersects the last segment of $\alpha^u(\Delta)$. If this intersection is at $\Delta = \Delta'$, then for all $\Delta < \Delta'$, $\beta^l(\Delta) < \alpha^u(\Delta)$ and $\sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\} \leq 0$ for all $\Delta \leq \Delta'$. This case is shown in Figure 28(b). Clearly, $r = r_\beta - s_\alpha$, $s = s_\beta - s_\alpha$, $q_1 = q_{1\beta} - q_{2\alpha}$ and $q_2 = q_{2\beta} - q_{2\alpha}$.

Case 3 is when the middle segment of $\beta^l(\Delta)$ intersects the middle segment of $\alpha^u(\Delta)$ (see Figure 28(c)). In this case, $\sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\}$ is made up of four linear segments. But we approximate it using $L(q_1, q_2, r, s)(\Delta)$,

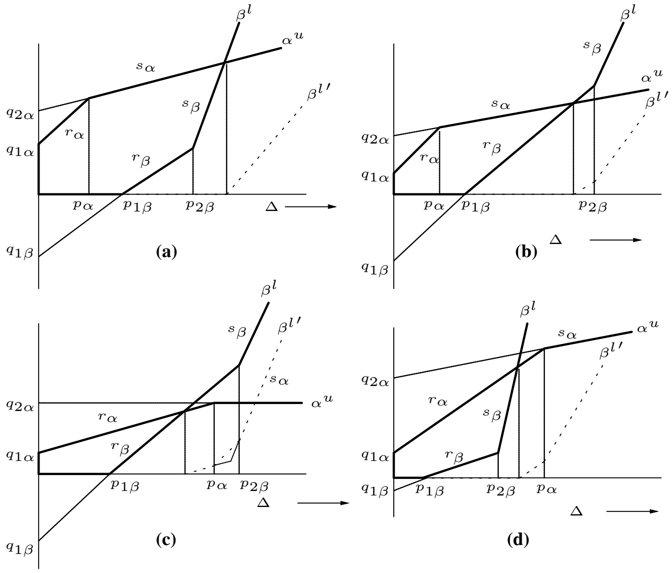

Fig. 28: Approximate remaining lower service curves. Figures (a), (b), (c) and (d) represent the cases 1, 2, 3 and 4 respectively in Theorem 14

which is made up of three segments. There can be two possible subcases, the first is when $p_{2\beta} \geq p_\alpha$ (as shown in Figure 28(c)), and the second is when $p_{2\beta} < p_\alpha$. If $\beta^l(\Delta)$ and $\alpha^u(\Delta)$ intersect at Δ', then the four segments that make up $\sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\}$ span the intervals $\Delta \in [0, \Delta')$, $[\Delta', p_\alpha)$, $[p_\alpha, p_{2\beta})$, $[p_{2\beta}, \infty)$ (as shown in Figure 28(c)) or $\Delta \in [0, \Delta')$, $[\Delta', p_{2\beta})$, $[p_{2\beta}, p_\alpha)$, $[p_\alpha, \infty)$ (in the case when $p_{2\beta} < p_\alpha$). To obtain $L(q_1, q_2, r, s)(\Delta)$, we neglect the segment of $\sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\}$ corresponding to the interval $[p_\alpha, p_{2\beta})$ in Figure 28(c) and the interval $[p_{2\beta}, p_\alpha)$ when $p_{2\beta} < p_\alpha$, and instead approximate this segment by the segments preceding and following it. It may be noted that $L(q_1, q_2, r, s)(\Delta)$ is a valid lower curve, since $L(q_1, q_2, r, s)(\Delta) \leq \sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\}$ for all $\Delta \geq 0$. Therefore, $r = r_\beta - r_\alpha$, $s = s_\beta - s_\alpha$, $q_1 = q_{1\beta} - q_{1\alpha}$, $q_2 = q_{2\beta} - q_{2\alpha}$.

Case 4 is when the last segment of $\beta^l(\Delta)$ intersects the middle segment of $\alpha^u(\Delta)$ (see Figure 28(d)). It can be seen that $r = s_\beta - r_\alpha$, $q_1 = q_{2\beta} - q_{1\alpha}$, and as before, $s = s_\beta - s_\alpha$ and $q_2 = q_{2\beta} - q_{2\alpha}$.

Lastly, if $\beta^l(\Delta) \leq \alpha^u(\Delta)$ for all $\Delta \geq 0$, then $\sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\} \leq 0$ for all $\Delta \geq 0$. Hence, $r = s = 0$ and $q_1 = q_2 = 0$. □

The approximations for all the remaining curves can be derived on the basis of similar techniques, and hence we omit them here.

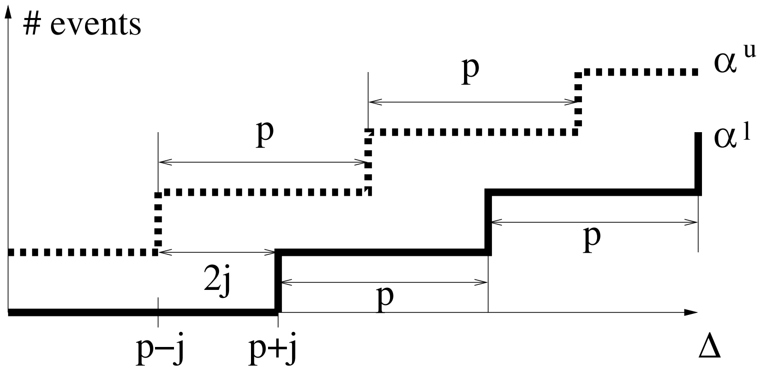

Fig. 29: The upper and lower arrival curves of the class of event streams with period p and jitter j.

4.5 Generalizing standard event models

Recall from Section 4.3 that $\alpha^l(\Delta)$ and $\alpha^u(\Delta)$ can be interpreted as the minimum and maximum number of events arriving within *any* time interval of length Δ, respectively. Standard event models can be represented in our model by an appropriate choice of α^l and α^u. For example, a periodic event stream with period p can be represented by an α^l and α^u, both of which are staircase functions of step width p and height 1, with $\alpha^l(\Delta) = 0$ for all $0 \leq \Delta < p$ and $\alpha^u(\Delta) = 1$ for all $0 < \Delta \leq p$. This is because within any time interval of length less than p, the minimum number of events that can be seen is zero, and within any time interval of length p^+, the minimum number of events that can be seen is equal to one. Similarly, the maximum number of events that can be seen within any time interval of length p and p^+ is one and two respectively.

Following the same reasoning, the class of event streams with period p and jitter j can be represented by an upper and a lower arrival curve of the form shown in Figure 29. Given any particular instance of such a periodic with jitter event stream, the corresponding upper and lower arrival curves would lie within the arrival curves shown in Figure 29, and therefore these curves represent the upper and lower bounds on the maximum and minimum number of events that can arrive within any time interval for any event stream with period p and jitter j. Alternatively, given the upper and the lower arrival curves of the class of event streams *periodic with jitter*, then it is possible to uniquely determine the period and the jitter values. Note that in Figure 29, if $j = 0$ then the upper and the lower arrival curves coincide and represent a purely periodic event stream with period p. Formally, these results can be stated as follows:

The upper and the lower arrival curves representing the entire class of event streams with period p and jitter j are unique.

Similar representations in terms of the upper and the lower arrival curves can be given for standard (abstract) event models like sporadic and periodic with bursts, or for other event streams with a known analytical behavior. At the same time, given any finite length arbitrary event trace (from measurements or from simulation) and a real number Δ, it is possible to determine the values of $\alpha^l(\Delta)$ and $\alpha^u(\Delta)$ corresponding to the event trace, by sliding a window of length Δ over the trace and recording the minimum and maximum number of events lying within the window respectively. The upper and the lower arrival curves corresponding to the trace can be determined by following this procedure for different values of Δ.

Now, in contrast to Equations (4.3) - (4.6) of Section 4.3 which were concerned with event streams that have a fixed starting time (say $t = 0$), the following equations hold for event streams which span over $t = -\infty$ to $t = +\infty$ and therefore accurately capture event models like periodic, sporadic, etc., which do not have any specific starting time. The results given in this section are based on these equations

$$\begin{aligned}
\alpha^{l\prime}(\Delta) &= \min\{\inf_{0\leq\mu\leq\Delta}\{\sup_{\lambda\geq 0}\{\alpha^l(\mu+\lambda)-\beta^u(\lambda)\} \\
&\qquad +\beta^l(\Delta-\mu)\},\beta^l(\Delta)\} && (4.10) \\
\alpha^{u\prime}(\Delta) &= \min\{\sup_{\lambda\geq 0}\{\inf_{0\leq\mu<\lambda+\Delta}\{\alpha^u(\mu)+ \\
&\qquad +\beta^u(\lambda+\Delta-\mu)\}-\beta^l(\lambda)\},\beta^u(\Delta)\} && (4.11) \\
\beta^{l\prime}(\Delta) &= \sup_{0\leq\lambda\leq\Delta}\{\beta^l(\lambda)-\alpha^u(\lambda)\} && (4.12) \\
\beta^{u\prime}(\Delta) &= \max\{\inf_{\lambda\geq\Delta}\{\beta^u(\lambda)-\alpha^l(\lambda)\},0\} && (4.13)
\end{aligned}$$

We now give two examples to show that in the case of heterogeneous system architectures, results from classical scheduling theory, that can be used to analyse standard event models (like periodic, sporadic, etc.), can also be derived within our framework. The work in [122] considered a number of examples of heterogeneous platform architectures involving standard event models and different scheduling strategies and answered various questions related to timing analysis using a compositional approach. By the use of some examples, it will be shown that similar and more general questions can be answered using the new unifying approach described in this chapter.

Ex. 1: *Consider a periodic event stream entering a resource which requires a maximum of* $e_{\max}$ *and a minimum of* $e_{\min}$ *time units to process an event. The outgoing (processed) event stream is still periodic, but has a jitter equal to* $e_{\max} - e_{\min}$.

Let t_0 be some sufficiently small time instance where all events that arrived before t_0 have been processed. Let $R[t_0, t)$ and $R'[t_0, t)$ denote the number of arrived and processed events in the interval $[t_0, t)$ for $t_0 < t$, respectively. Then we can derive $R[t_0, t - e_{max}) \leq R'[t_0, t) \leq R[t_0, t - e_{min})$ if $t_0 < t - e_{max}$. Using these inequalities, we find for $s < t$ the relation $R'[s, t) \leq R[t_0, t - e_{min}) - R[t_0, s - e_{max}) = R[s - e_{max}, t - e_{min}) \leq \alpha^u((t - s) + (e_{max} - e_{min}))$. In a similar way, we have $R'[s, t) \geq R[t_0, t - e_{max}) - R[t_0, s - e_{min}) = R[s - e_{min}, t - e_{max}) \leq \alpha^l((t - s) - (e_{max} - e_{min}))$ for $t - s > e_{max} - e_{min}$.

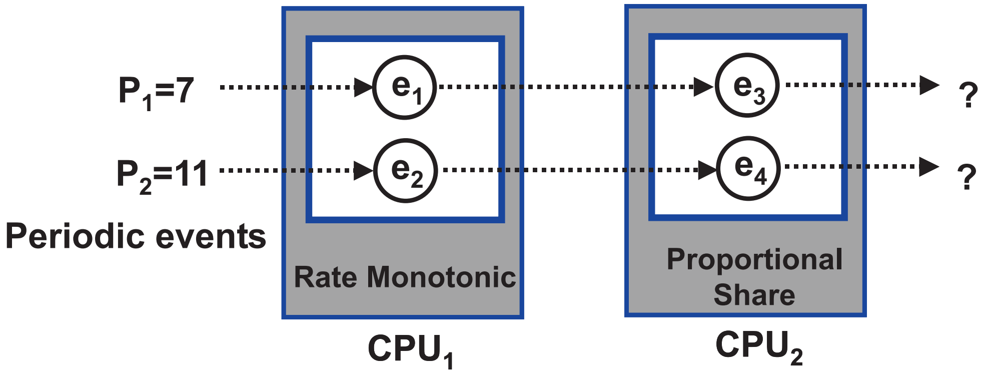

Fig. 30: The system described in Example 2.

As a first result, we find the feasible lower and upper curves of the processed events as $\alpha^{u\prime}(\Delta) = \alpha^u(\Delta + (e_{max} - e_{min}))$ for $\Delta > 0$ and $\alpha^{l\prime}(\Delta) = \alpha^l(\Delta - (e_{max} - e_{min}))$ for $\Delta > e_{max} - e_{min}$ and $\alpha^{l\prime}(\Delta) = 0$ otherwise.

Hence, the number of events that can be seen at the output within any time interval of length Δ is greater than or equal to the number of events that can be seen at the input over any time interval of length $\Delta - (e_{\max} - e_{\min})$, and is less than or equal to the number of events that can be seen at the input within any time interval of length $\Delta + (e_{\max} - e_{\min})$. This implies that the jitter of the output event stream increases by $(e_{\max} - e_{\min})$ over the jitter of the input event stream. If the input stream is purely periodic with a period p, then the output stream is periodic with period p and jitter equal to $(e_{\max} - e_{\min})$.

Ex. 2: *A system consists of two processors CPU_1 and CPU_2, on each of which two processes are implemented, as shown in Figure 30. Two purely periodic event streams 1 and 2, with periods $p_1 = 7$ and $p_2 = 11$ respectively are processed by the two processes implemented on CPU_1. The per event processing time for both the event streams is equal to 2. CPU_1 schedules the two processes processing streams 1 and 2 according to rate monotonic scheduling, and therefore stream 1 has higher priority over stream 2. The two outgoing, processed event streams are then processed by the two processes implemented on CPU_2, where the per event processing time is again equal to 2. CPU_2 implements proportional share scheduling and gives equal processor share to both the processes. Both CPU_1 and CPU_2 implement preemptive scheduling. What are the characteristics of the two processed event streams coming out of CPU_2?*

We use the arrival curves of the input event streams entering CPU_1 and from them compute the arrival curves of the final processed event streams coming out of CPU_2. These are then used to deduce the timing behaviour of the processed event streams. Figure 31 shows the scheduling network corresponding to the system. The entire processing capability of CPU_1 is available to stream 1 since this has the higher priority. This is represented by $\beta^u = \beta^l$, both being straight lines of slope 1 passing through the origin. Figure 32 shows the arrival curves

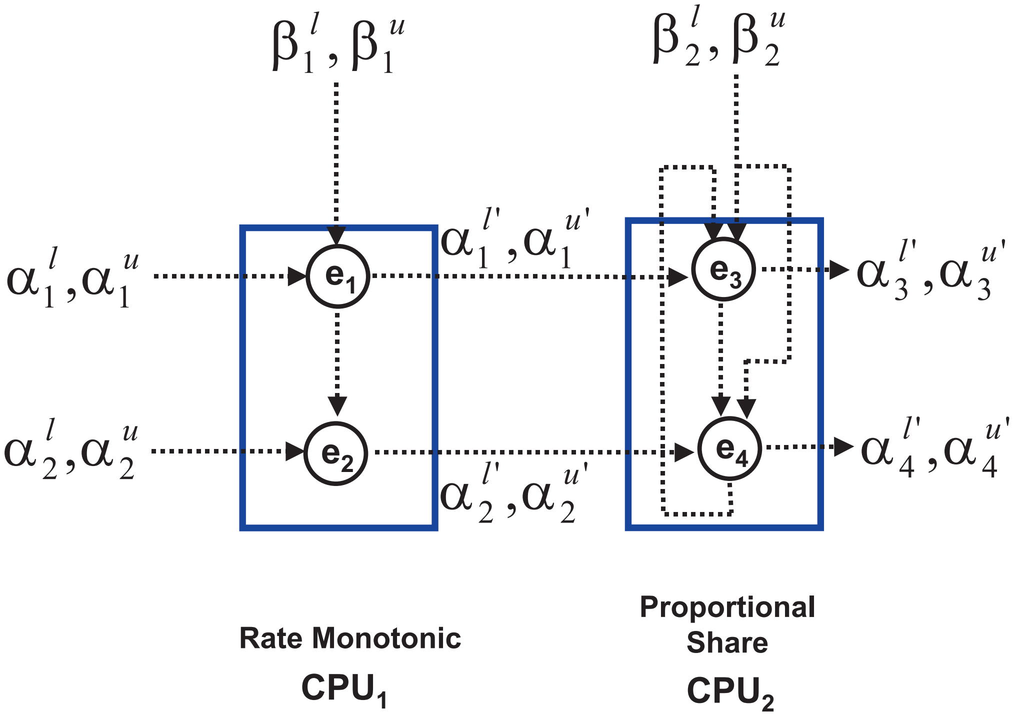

Fig. 31: The scheduling network for the system described in Example 2.

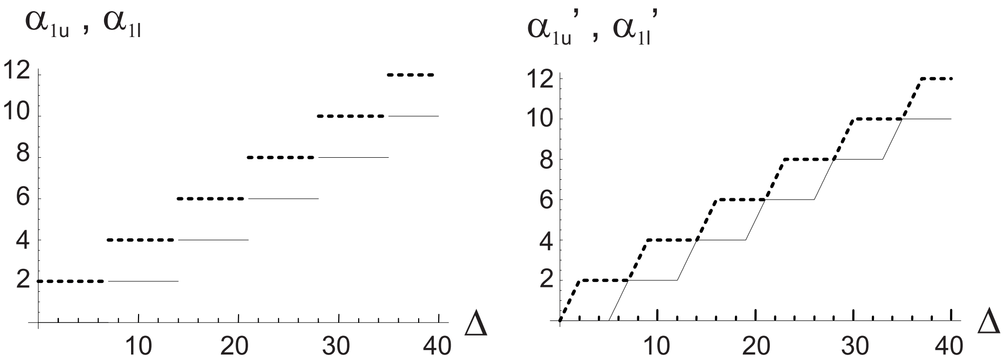

Fig. 32: The upper and lower arrival curves of the incoming event stream 1 and the arrival curves of the processed stream coming out of CPU_1 (dotted line show the upper curve and the solid line shows the lower curve).

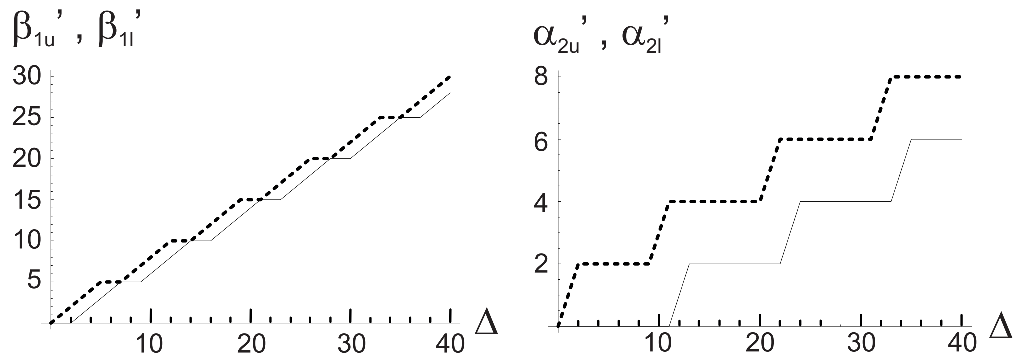

Fig. 33: The service curves used to process stream 2 in CPU_1 and the arrival curves of the processed stream coming out of CPU_1.

of the stream 1 and those of the processed stream. Note that the processed stream is still periodic with period 7. In Figure 32, the arrival curves of the input event stream represent the discrete stream, but since the Equations (4.10–4.13) hold for continuous streams, to interpret the right hand figure in Figure 32 as a discrete stream, a *floor* function should be applied to the lower curve and a *ceiling* function to the upper curve.

The remaining processing capability of CPU_1 that is available to stream 2 can be obtained by using Equations (4.12) and (4.13). These resulting service curves and the arrival curves of the processed stream are shown in Figure 33. The arrival curve of stream 2 is shown on the left hand side of Figure 35. As can be seen from this figure, the processed stream is still periodic with period 11, but now has a jitter smaller than or equal to 2.

In the case of CPU_2, β_2^u and β_2^l represent the total unloaded processor capacity (see Figure 31) and are given by straight lines of slope 1 passing through the origin. Because of the proportional share scheduling, both the incoming streams into CPU_2 are guaranteed at least 50% of the available resource. But if one stream does not fully use its allocated resources, then the resulting leftover is available to the other stream. Therefore, the upper and lower service curves for stream 3 (i.e. the processed stream 1) are equal to $\beta^u(\Delta) = 0.5 \cdot \beta_2^u(\Delta) + \max\{\inf_{\lambda \geq \Delta}\{0.5 \cdot \beta_2^u(\lambda) - \alpha_4^l(\lambda)\}, 0\}$ and $\beta^l(\Delta) = 0.5 \cdot \beta_2^l(\Delta) + \sup_{0 \leq \lambda \leq \Delta}\{0.5 \cdot \beta_2^l(\lambda) - \alpha_4^u(\lambda)\}$ respectively. Here, α_4^l and α_4^u (which are equal to $\alpha_2^{l\prime}$ and $\alpha_2^{l\prime}$, respectively) are the arrival curves of stream 4. The service curves available to stream 4 can similarly be computed from β_2^u, β_2^l, α_3^l ($= \alpha_1^{l\prime}$) and α_3^u ($= \alpha_1^{u\prime}$).

Based on these service curves, the arrival curves of the processed streams 3 and 4 (by CPU_2) are given in (the right hand of) Figures 34 and 35. From these curves, it can be deduced that the processed stream 1 (after passing through CPU_1 and CPU_2) has period 7 and a jitter smaller than or equal to 2 and the processed stream 2 (after passing CPU_1 and CPU_2) has the period 11 and jitter smaller than or equal to 4. These values exactly conform to those that can be

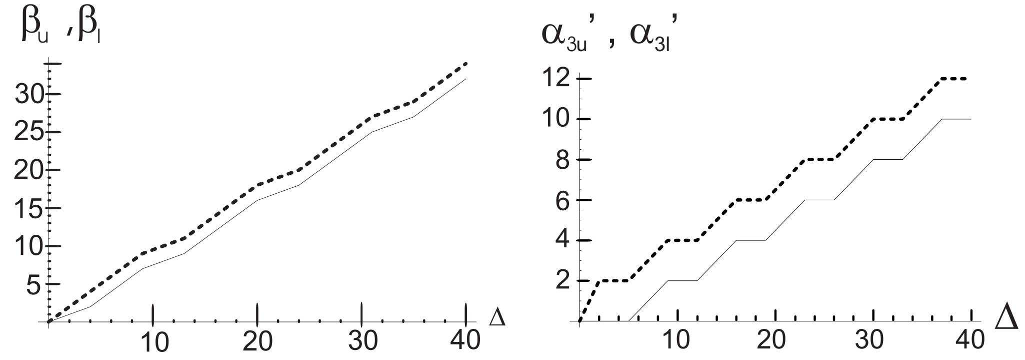

Fig. 34: The service curves used to process stream 1 coming out of CPU_1 (indicated as stream 3) and the arrival curves of the processed stream (by CPU_2).

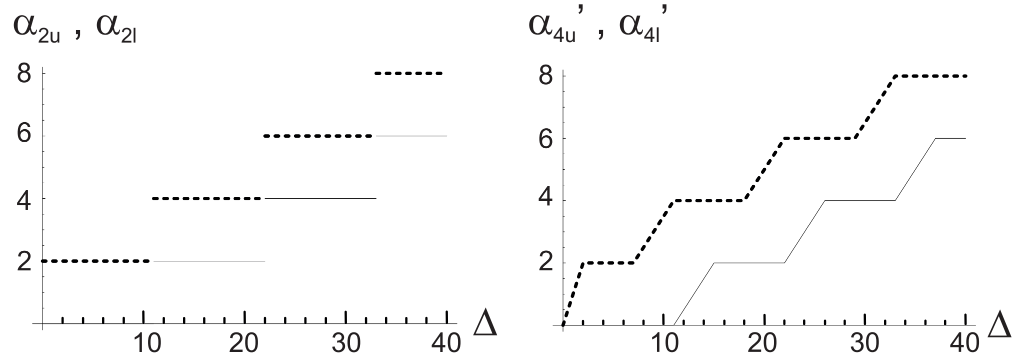

Fig. 35: The arrival curves of the incoming event stream 2, and those of the finally processed stream coming out of CPU_2 (i.e. after being processed at CPU_1 and CPU_2).

obtained using classical scheduling theoretic results.

4.6 The simulation setup

Even in cases where analytical models exist, performance evaluation of processor architectures using simulation still continues to be the most widely used procedure. This is primarily motivated by the accuracy of the results that can be obtained using simulation, and the second reason being flexibility. In many cases analytical models can not capture all the aspects of an architecture and are often restricted to one particular level of abstraction.

In this section we outline the methodology for model composition and performance evaluation of network processor architectures based on simulation, which forms the basis for comparing the results obtained from the analytical framework presented in the last two sections. This section is based on the work presented in [155] and more detailed explanations concerning the models can be

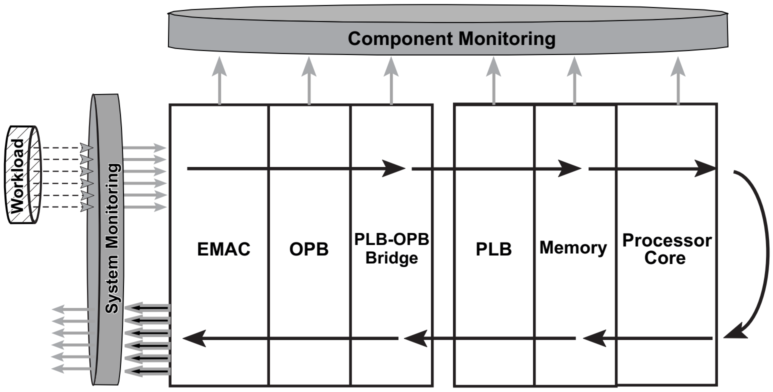

Fig. 36: System evaluation and component evaluation of a network processor architecture.

found there. Here the primary goal is to illustrate the abstraction level at which the different components of the architecture are modeled. Section 4.7 presents the results obtained by evaluating a reference network processor architecture using this simulation method along with the results obtained by the analytical model on the same architecture.

4.6.1 Modeling environment and software organization

The overall approach is based on using a library of reusable component models written in SystemC [73, 140]. These include models for buses, different interfaces, processor cores, etc. Each of these models can be an abstract match of an already existing hardware component which can be found from a core library [80], or can also be a model of a new component which does not exist yet. In an architecture composition step the selected component models are combined into a system model which is then evaluated by simulation. The workload used to drive the simulation can either be synthetically generated, or be obtained from real traffic traces. During simulation, the model execution performance data can be collected which can then be evaluated later.

It is not necessary that every component model is implemented on the same level of abstraction. But all models are implemented in the form of a *black-box* having well defined abstract interfaces and allow for component-specific local refinements. This supports easy exchangeability among different models of the same component. Every component model is separated into two layers—a so called abstract functional layer and a data collection layer. The functional layer describes the functional behavior of a component and defines the component interfaces, while the data collection layer exclusively deals with gathering statistics which are used to evaluate different performance metrics. This separation enables independent and selective refinements on the functional layer and also flexible customization of the data collection mechanisms.

The performance evaluation done using the data collected during simulation can be distinguished into either a component level evaluation or a system evaluation. These are illustrated in Figure 36 (note that this shows the organization of

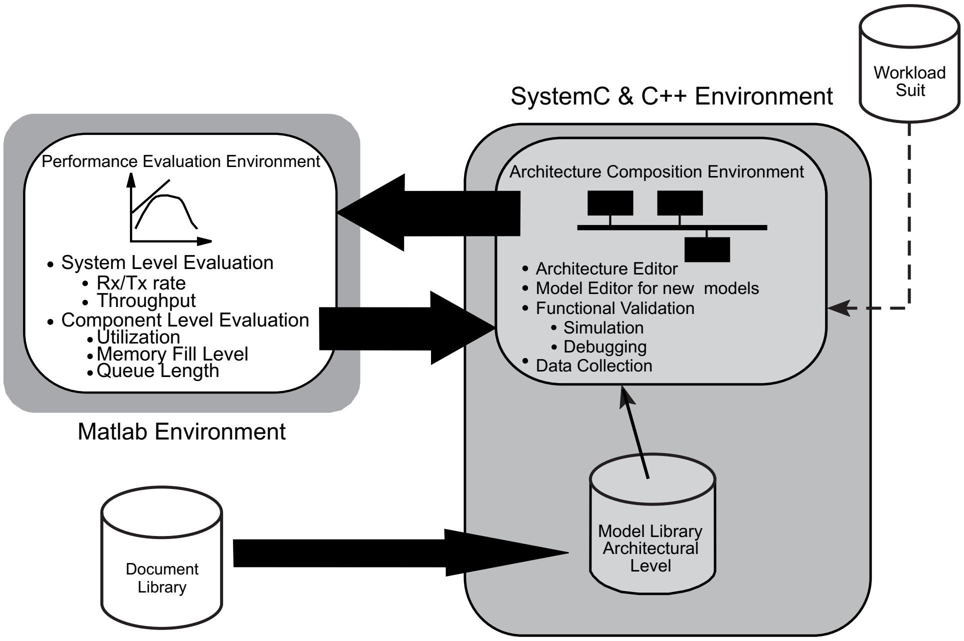

Fig. 37: Overview of the simulation framework that is used for comparing the results obtained by the analytical method.

the framework, the models of the different architectural components might vary in their level of abstraction and details). The component evaluation is based on the data collected through the data collection layer of each component model. Examples of such evaluation metrics can be memory fill levels, bus rearbitration counts, and the load on the different components. System specific aspects of an architecture like the overall system throughput, or end-to-end packet delays are evaluated using system evaluation mechanisms. In contrast to the component evaluation approach, the data in this case is not gathered within a specific component but is collected using the workload traveling through the entire system.

An overview of the entire simulation framework is given in Figure 37. Based on specification documents of already existing components, a set of abstract models are created and combined with component- and system-specific data collection metrics. Such models constitute, what is referred to in the figure as the “Model Library”. The architectural models from this library can be composed together and then simulated on a workload.

4.6.2 Component Modeling

In this section we briefly describe how each component of the reference network processor architecture that is evaluated in Section 4.7 is modeled in the simulation. This would enable a meaningful comparison between the results obtained by simulation and those obtained from the analytical model. The models usually available from core libraries are in the form of hard or soft cores, whereas SystemC models used in the simulation were created for the work done in [155]. The discussion below refers to these SystemC models.

Bus models: The bus models used in the reference architecture are based on the CoreConnect bus architecture [81], designed to facilitate core based designs. CoreConnect involves three buses: The Processor Local Bus (PLB) is for interconnecting high performance cores with high bandwidth and low latency demands, such as CPU cores, memory controllers and PCI bridges. The On-Chip Peripheral Bus (OPB) hosts lower data rate peripherals such as serial and parallel ports and UARTs. The PLB is fully synchronous, has decoupled address, read and write data lines and transfers on each data line are pipelined. Different masters may be active on the PLB address, read and write data lines and access to the PLB is granted through a central arbitration mechanism that allow masters to compete for bus ownership.

The models used for the PLB and the OPB are both cycle accurate. Both these models do not transfer any data nor consider any address, but model the signal interchanging according to the bus protocol.

There is a third bus in CoreConnect, which we do not use in our study. This is called the Device Control Register (DCR) bus, and is used for reading and writing low performance status and configuration registers.

The features of the PLB that are modeled for our study include the arbitration mechanism, the decoupled address and read and write lines, a fixed length burst protocol, and a slave enforced arbitration mechanism. The OPB is much less sophisticated compared to the PLB and most of its features are modeled. For both the buses, the arbitration algorithm uses a round robin strategy among the requesting masters on a given priority level, and typically there are four priority levels.

Ethernet core (EMAC) model: The Ethernet core or EMAC is a generic implementation of the Ethernet media access control (MAC) protocol, supporting both half-duplex (CSMA/CD) and full duplex operations for Ethernet, Fast Ethernet and Gigabit-Ethernet. The EMAC has two large FIFOs to buffer packets, and two OPB interfaces—one for providing access to its configurations and status registers, and the other is a bidirectional interface to transfer data to and from the PLB-OPB bridge.

The model of the EMAC only contains receiving and transmitting channels, where a receive channel can be considered as an input port and a transmit channel as an output port.

The set of receiving channels constitutes the traffic input to the network processor. Each one reads packet information from a real traffic trace at a parameterizable rate. Within a receive channel there are two threads of activity. The first one reads the input packet traces, and writes each resulting packet into a FIFO. The second thread implements the communication protocol with the PLB-OPB bridge and transfers packet to memory as long as the FIFO is not empty. The transmit path consists only of a *transmit packet* thread, which is active as long as packets are waiting to be transferred for the appropriate port.

PLB-OPB bridge model: The PLB-OPB Bridge is a specialized module which combines pure bridge functionality with DMA capability, and it can effectively handle packet transfer overheads. An EMAC communicates with a PLB through the PLB-OPB bridge. Since as an OPB slave, the EMAC can not inform the bridge of its willingness to transfer a packet, the EMAC to PLB-OPB bridge interface has *sideband* signals to meet this purpose. These do not form a part of the OPB bus, and nearly all signals are driven by the EMAC and sampled by the bridge.

The PLB-OPB bridge is modeled as two independent paths, receive and transmit, and both offer the bridge functionality and arbitrate among active channels. Each path implements the PLB-OPB bridge to EMAC communication protocol by driving the required sideband signals and accessing buses. Bus accesses are concurrent and therefore both paths can contend for their access, especially on the OPB.

Memory model: The memory is accessed via the PLB and can either be on-chip or off-chip. It is modeled in a high level way, where only parameters like average or worst case transfer latency are considered.

Software application and timing: A simple high-level model of software application is used. It primarily consists of the following. For each packet the software can cause a pure delay without generating any PLB load, representing processing time in the CPU. Second, there can be a PLB load generated by the software (for example, this might consist of reading and writing packets by the CPU to and from the memory). Lastly, the timing model is based on using the PLB clock as the system time.

4.7 A comparative study

This section presents the main result of this chapter—a comparison of the performance evaluation data obtained by the analytical framework presented in Sections 4.3 and 4.4 on a reference network processor architecture, with the results obtained by detailed simulations based on the models discussed in Section 4.6. This comparative study is based on the assumption that there is a high confidence in the simulation results. We do not compare the results obtained by either the analytical method or the simulations with any real hardware implementation because of two reasons: (i) During a design phase an actual hardware does not exist, and the best one can do is to validate the results obtained from an analytical model against those obtained using simulation, and vice versa, (ii) Simulations are widely used in practice, and assuming that the processor model being simulated is accurate and detailed enough, the results are expected to match the real hardware with high confidence.

Our choice of the reference architecture which is the basis of the comparison is a hypothetical system-level model of a network processor which can be matched by many existing network processors (such as the family of processors described in [119]). The architectural components modeled in this study are from an existing core library [80]. Since the particular system-level model we study here is sufficiently general, we believe that the conclusions based on the results obtained from this study would hold for many other models. This enables us in making the general claim that it is possible (and more appropriate in terms of the benefits in running time) to use analytical modeling frameworks during the early stages of architecture design space exploration in the context of network processors, in order to tackle the design complexity.

We evaluate the reference architecture using three different performance metrics: (i) The line speed or the end-to-end throughput that can be supported by the architecture. This is measured using the utilization of the different components of the architecture and hence also identifies the component which acts as the bottleneck. During a design space exploration, identifying the utilization of the different components goes beyond measuring the overall throughput of the system because a designer in generally interested in identifying whether all the components in the architecture have a moderately high utilization at the maximum supported line speed, or whether it is one single component that acts as a bottleneck. (ii) The maximum end-to-end delay that is experienced by packets from the different flows being processed by the architecture. (iii) The total on-chip buffer/memory requirements, or in other words, the on-chip memory fill level. The results of the analytical framework should be judged on the basis of how closely the data related to these performance metrics for the reference architecture match those obtained using simulation. Rather than absolute values, it is more interesting to analyse the behaviour of the architecture (for example with increasing line speeds, or increasing packet sizes for the same line speed), and see if the same conclusions can be drawn from both the evaluation techniques. The second axis for our comparisons is the time it takes to compute the evaluation data by the analytical framework, compared to the simulation time required to generate this data.

4.7.1 Reference architecture and parameters

The system-level model of a network processor that is used for this study is shown in Figure 24. The different actions that are executed while each packet travels through this architecture, and the order in which they are executed is given in Table 3. The model effectively deals with the communication subsystem of the architecture, and the software application running on the CPU core (indicated by PPC - PowerPC) is modeled as simply performing two reads from the SDRAM and one write to the SDRAM for every processed packet. The amount of data read or written (and hence the traffic generated on the PLB), however, depends on the packet size and this is appropriately modeled.

As seen in Figure 24, the architecture is composed of two Ethernet media access controllers (EMACs), a slow on-chip peripheral bus (the OPB), a fast processor local bus (the PLB) consisting of separate read and write lines, a PLB-OPB bridge, a SDRAM and a processor core (PPC). Each EMAC consists of one receive and one transmit channel, and is capable of reading packets at parameterizable input rates.

In the simulation, the entire path of a packet through the modeled architecture can be described as follows. First a receive channel of an EMAC reads a packet from a file containing the packet traces (only packet lengths are used, and all packets arrive back-to-back with a fixed interframe gap; this is described in further details later), and allocates a packet record which contains the packet length and the source EMAC identification. This packet record models the packet inside the processor architecture and generates a load equivalent to the size of the packet. The channel then requests service to the PLB-OPB bridge via a sideband signal, which is served following a bridge internal arbitration procedure. The PLB-OPB bridge fetches a *buffer descriptor* for the packet (which is a data structure containing a memory address in the SDRAM, where the received packet is to be stored). This fetching operation involves the SDRAM and generates traffic on the PLB read bus, equal to the size of the buffer descriptor. Following this, the received packet is stored in the SDRAM at the location specified by the buffer descriptor. This involves the packet traversing through the OPB to the PLB-OPB bridge, and then through the PLB write bus to the SDRAM. This generates a load equal to the size of the packet, on both the buses. Since data on the PLB is sent in bursts, the PLB-OPB bridge schedules a PLB transfer only when sufficient data is gathered. As the EMAC channel is served over and over again, the packet is written part by part into the SDRAM. After the packet transfer is complete, the bridge informs the EMAC receive channel via a sideband signal, and also notifies the application software running on the processor core (again by a sideband signal) that the packet is now available in the memory. It is then processed by the software as soon as the processor becomes available. This processing involves a buffer descriptor transfer from the SDRAM to the processor core via the PLB read bus, followed by a packet header transfer, again from the SDRAM to the processor core via the PLB read bus. The packet header is then processed in the processor core (for example implementing some lookup operation) and written back into the SDRAM over the PLB write bus.

After the completion of this processing, the software notifies the bridge (via a sideband signal) that the packet is now processed and is ready to be sent out through the chosen transmit channel of the EMAC. As in the receive path of the packet, the bridge gets the buffer descriptor of the packet from the SDRAM via the PLB read bus, and then the packet traverses over the PLB read bus and the OPB to an EMAC transmit channel. After the completion of the packet transfer the EMAC notifies the bridge via a sideband signal, which then reads certain status information and releases the buffer descriptors.

This entire process happens concurrently for two packet flows entering through the two EMACs of the architecture. All the components involved also work concurrently and the two buses (the PLB and the OPB) use first-come-first-serve as a bus arbitration policy. The main complexity in the analysis of this system is due to concurrent operation of the different components. Hence it is non-trivial to evaluate how the system behaves with increasing line speeds, variations in packet sizes, and what is the maximum line speed that it can support without packet dropping.

Parameters: As already mentioned, the EMAC can read packets at different input line speeds. The line speeds used for the evaluation range from 100 Mbps to 400 Mbps, the former representing a nominal load situation and the later a loaded situation.

The OPB modeled has a width of 32 bits and a frequency of 66.5 MHz. The read and the write data paths of the PLB are of 128 bits and operate at 133 MHz. The size of a PLB burst is limited to a maximum of 64 Bytes. Therefore, the PLB-OPB bridge gathers up to 64 Bytes (which is only one OPB burst transfer) before scheduling the PLB transfer. There are two different kinds of buffer descriptors, small and large ones. The small buffer descriptors refer to memory locations/buffers with 64 Bytes of size, while the large ones refer to buffers with a size of 1472 Bytes. As a consequence, 64 Byte packets require only one small buffer descriptor and packets larger than 64 Bytes require an additional large buffer descriptor. Both *small* and *large* buffer descriptors are of size 64 Bytes each. Therefore, the traffic generated by a packet on any of the buses depends not only on its own size, but also on the buffer descriptors associated with it. All packets and the buffer descriptors reside in the SDRAM described above.

4.7.2 Evaluation method and comparisons

The reference architecture described above is evaluated using simulation and the analytical framework using two different workload types—synthetic traces with same-sized packets, and real traces from NLANR [102]. For the synthetic traces, packet sizes of 64, 128, 512, 1024, 1280 and 1500 are used. The real traces are used only to exploit the impact of real world packet size distributions on the system performance. They are time compressed and adjusted and only the packet sizes are retained. Therefore, in both the cases packets arrive back-to-back (to exert the maximum stress on the architecture) with an interframe gap equal to 20 Bytes.

The overall scheme for comparing the results obtained using the analytical framework with those obtained from simulation is shown in Figure 38. The different components of the architecture are modeled in either SystemC in the simulation based evaluation, or analytically using the model presented in Section 4.3. To compute the required parameters of an analytical component model (such as the transfer time of a single packet over an unloaded bus), either simulation results are used or data sheets of the component are used. The component models are then composed together (using the methods described in Section 4.4 in the case of the analytical framework, and using standard SystemC compo-

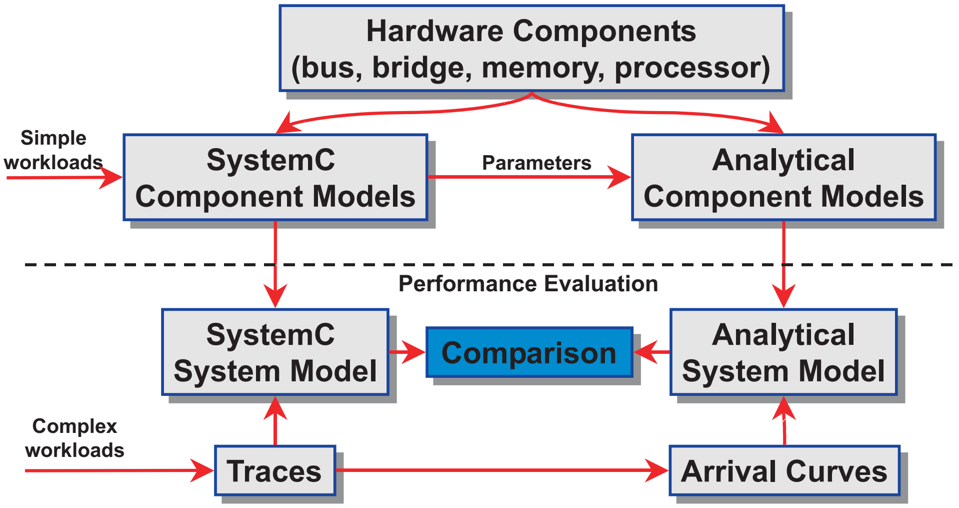

Fig. 38: The overall scheme for comparing the results from the analytical framework with those obtained by simulation.

sition techniques in the case of simulation) to obtain a system model of the architecture.

Recall that the analytical model considered here does not use real packet traces, but instead uses arrival curves modeling the traces in terms of their maximum packet size, burstiness, and long term arrival rate. These parameters were extracted from the traces as shown in Figure 39 and fed into the model for evaluation. For the upper arrival curve, the maximum number of bytes that can arrive (at the network processor) at any time instant is given by the largest sized packet, the short-term burst rate is given by the maximum number of largest sized packets that can be seen occurring back-to-back in the trace, and the long-term arrival rate is given by the total length (in Bytes) of a trace divided by the time interval over which all the packets in this trace arrive.

Similarly, for the lower arrival curve, the maximum time interval over which no traffic can arrive is equal to the inter-frame gap in the trace (equal to 20 Bytes), the bound on the minimum number of packets that can arrive over a time interval is given by the maximum number of minimum sized packets occurring back-to-back in the trace, and the long-term arrival rate is equal to that in the upper arrival curve.

Given any packet trace, arrival curves such as those shown in Figure 39 can be derived from the trace and they capture the traffic arrival pattern given by the trace. Note that here we restrict each arrival to be made up of a combination of three line segments in order to simplify the computations involving these curves. However, in general they can be arbitrarily complex to capture the exact details of a trace (albeit, at the cost of increasing the computational complexity). As mentioned in Section 4.4 the analytical model composes the different component models, resulting in a *scheduling network*. For the architecture we study here (Figure 24), such scheduling network is given in Figure 25.

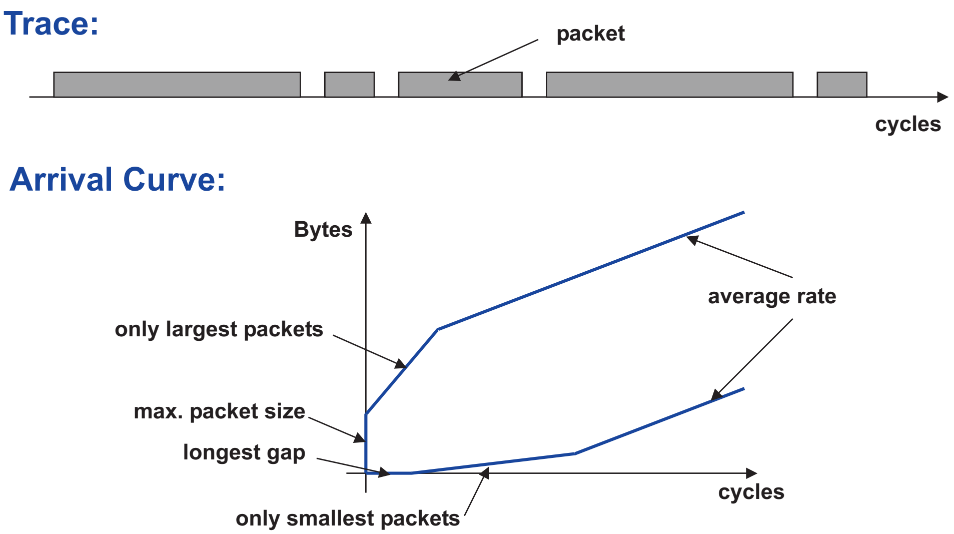

Fig. 39: Obtaining arrival curves from packet traces in the analytical framework.

4.7.3 Evaluation results

Table 4 gives the utilization values of the three buses (the OPB, and the PLB read and write bus) when the model is fed with synthetic traces consisting of fixed sized packets. Here, six different packet sizes have been used, from 64 Bytes to 1500 Bytes. For each packet trace and bus combination, the table compares the results obtained from the analytical method with those resulting out of simulation for different line speeds. To give an impression of how the utilization of the different buses increase with the line speed for the same packet size, in Figure 40 we plot the utilization values for the trace containing 512 Byte sized packets. As can be seen from Table 4, the results for the other traces are very similar, and hence we do not plot them.

There are two things to be noted from these values. First, with increasing line speeds the utilization of the different buses also increase, and as expected, this increase is proportional to the increase in the line speed. Second, the results from the analytical method and the simulation match very well for the utilization. In Figure 40, for each line speed there are three bars, each corresponding to the OPB, the PLB read and the PLB write bus. It may be noted from the figure that the maximum line speed that this architecture can sustain is in the range of 400 Mbps and the OPB acts as a bottleneck (as most of the traffic is generated on it).

In Figure 41 we show how the utilization values of the PLB read bus for fixed line speeds, as the packet size is increased. For a fixed line speed, as the packet size is increased, the component of the utilization that comes from the packet traversal increases, since there is less total interframe gap in the whole trace (assuming that the trace size in bytes remains the same). This is because the number of packets in the trace decrease. However, because of this the number of buffer descriptor and packet header traversals also decrease and therefore

Packet Size 64 Bytes

	OPB		PLB read		PLB write	
	AnM	Sim	AnM	Sim	AnM	Sim
100 Mbps	18	18	14	13	6	5
150 Mbps	27	28	20	19	9	8
200 Mbps	36	37	27	25	12	10
250 Mbps	45	46	34	31	15	13
300 Mbps	54	55	41	37	17	15
350 Mbps	63	65	48	40	20	17
400 Mbps	72	76	55	47	23	20

Packet Size 128 Bytes

	OPB		PLB read		PLB write	
	AnM	Sim	AnM	Sim	AnM	Sim
100 Mbps	19	19	15	14	5	5
150 Mbps	29	28	22	21	7	7
200 Mbps	38	38	30	28	10	10
250 Mbps	48	47	37	35	12	12
300 Mbps	57	56	45	42	15	15
350 Mbps	69	66	52	48	17	16
400 Mbps	79	75	59	53	20	18

Packet Size 512 Bytes

	OPB		PLB read		PLB write	
	AnM	Sim	AnM	Sim	AnM	Sim
100 Mbps	20	20	7	7	3	3
150 Mbps	30	29	11	11	5	4
200 Mbps	40	39	15	15	7	6
250 Mbps	50	49	19	19	8	7
300 Mbps	60	59	22	22	10	8
350 Mbps	71	69	26	26	12	10
400 Mbps	82	79	30	30	13	11

Packet Size 1024 Bytes

	OPB		PLB read		PLB write	
	AnM	Sim	AnM	Sim	AnM	Sim
100 Mbps	20	20	6	6	3	2
150 Mbps	30	30	9	9	4	4
200 Mbps	40	40	12	12	6	5
250 Mbps	50	50	15	15	7	6
300 Mbps	60	59	18	18	9	7
350 Mbps	71	69	21	21	10	8
400 Mbps	83	79	25	24	12	9

Packet Size 1280 Bytes

	OPB		PLB read		PLB write	
	AnM	Sim	AnM	Sim	AnM	Sim
100 Mbps	20	20	6	6	3	2
150 Mbps	30	30	9	9	4	4
200 Mbps	40	40	12	12	6	5
250 Mbps	50	50	15	15	7	6
300 Mbps	60	60	18	18	9	7
350 Mbps	71	69	20	21	10	8
400 Mbps	83	79	23	24	12	9

Packet Size 1500 Bytes

	OPB		PLB read		PLB write	
	AnM	Sim	AnM	Sim	AnM	Sim
100 Mbps	20	20	6	6	3	2
150 Mbps	30	30	9	9	4	4
200 Mbps	40	40	12	12	6	5
250 Mbps	50	50	14	15	7	6
300 Mbps	61	60	17	18	9	7
350 Mbps	72	70	20	21	10	8
400 Mbps	83	80	23	24	12	9

Tab. 4: The utilization values of the OPB, and the PLB read and write buses, when the model is fed with six different synthetic traces consisting of fixed sized packets (ranging from 64 to 1500 Bytes). For each trace and for each bus, the first column (marked as AnM - analytical method) gives the results obtained using the analytical model, and the second column (marked as Sim) gives the results obtained by simulation.

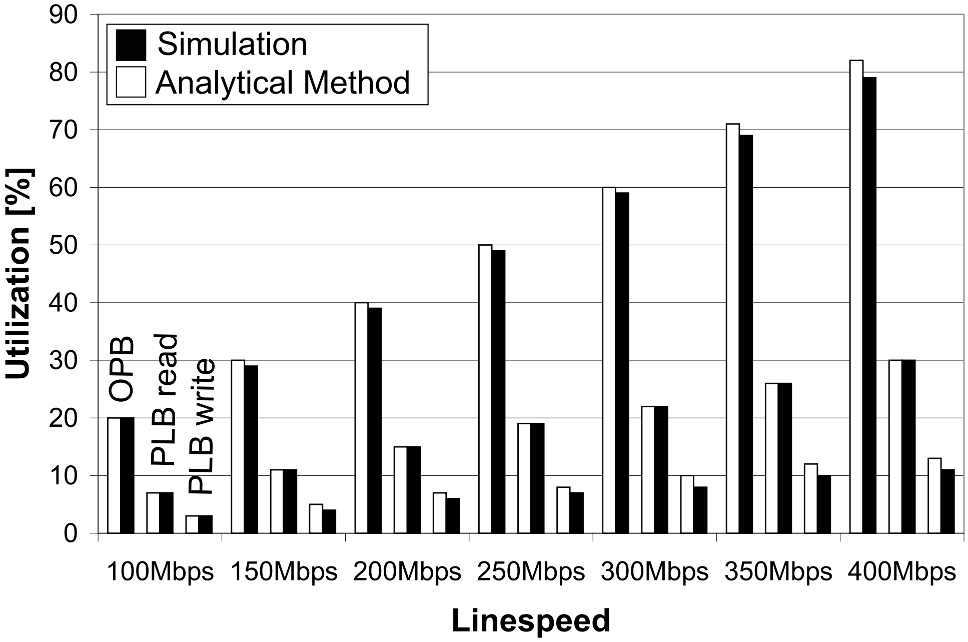

Fig. 40: Utilization values for different line speeds for the trace containing 512 byte packets. In this bar graph, for each line speed, the first bar indicates the utilization of the OPB, the second bar shows the utilization of the PLB read bus and the third bar corresponds to the PLB write bus. For each bus, the white bar gives the result computed by the analytical method, and the black bar gives the result obtained from simulation.

the component of the bus utilization that comes from these also decrease. These effects can be seen in Figure 41. As the packet size is doubled from 64 to 128 Bytes, the first component mentioned above plays a dominating role and hence the utilization slightly increases. Thereafter, the effect of the second component dominates and the utilization falls, and then remains almost constant since there is no significant change in the number of packets as the packet size is increased from 1024 to 1280 Bytes and then from 1280 to 1500 Bytes. The same results for the OPB is shown in Figure 42. However, in this case the utilization increases with increasing packet size because these are no packet header or buffer descriptor transfers over this bus.

It is to be noted that the match between the analytical results and the simulation is always close enough to deduce the above conclusions from the analytical results itself (with significant savings in evaluation time).

For the fixed size packet traces, we do not consider the end-to-end packet latencies and the memory fill levels, since for all low load situations they remain constant and do not depend on the input line speed.

Next we consider the results generated by real traces obtained from NLANR [102]. Use three different traces—FL (traces from a number of Florida universities), SDC (traces collected from the San Diego Supercomputer Center) and TAU (traces from the Tel Aviv University). Each trace is made up of traffic patterns for two different lines and these are fed into the two EMACs in our

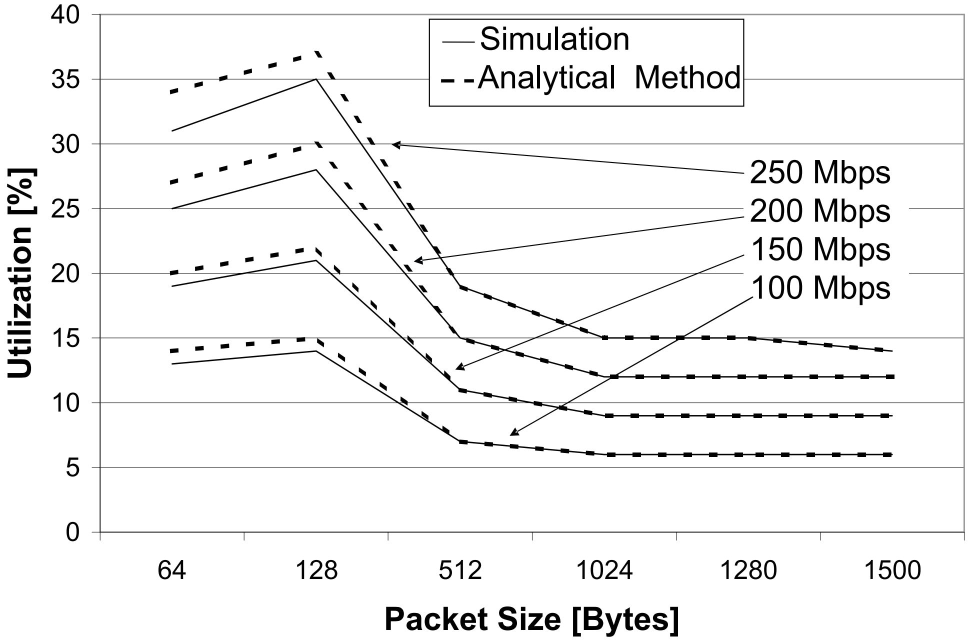

Fig. 41: The variation of the PLB read bus utilization with increasing packet size for four different line speeds.

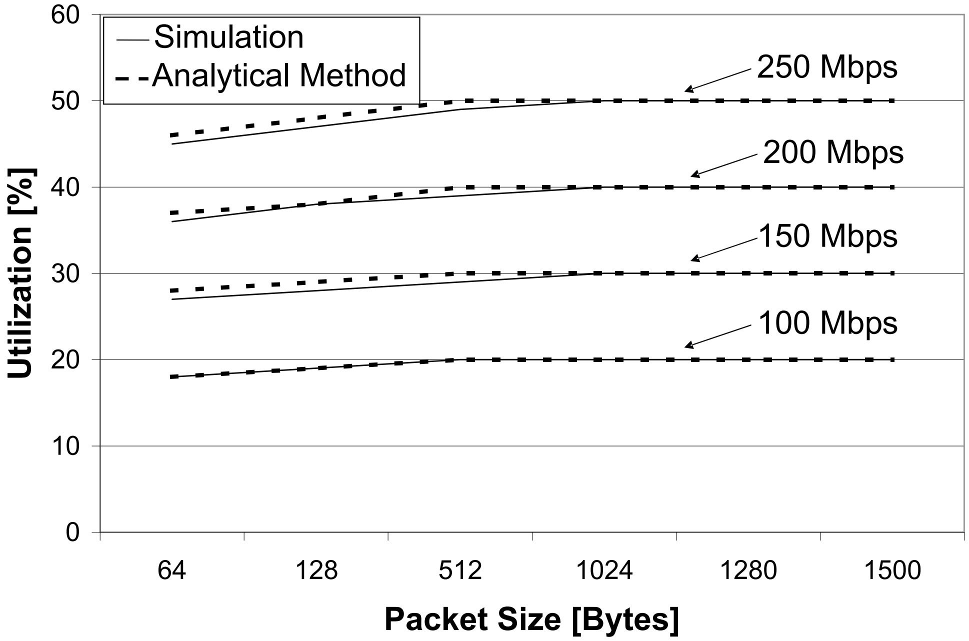

Fig. 42: The variation of the OPB utilization with increasing packet size for four different line speeds.

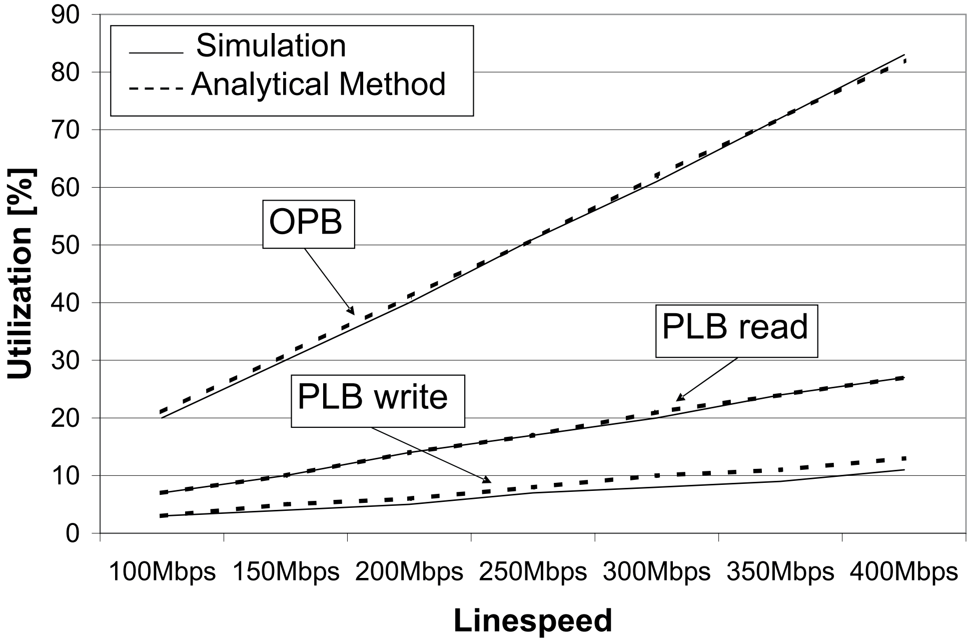

Fig. 43: The utilization of the OPB and the PLB read and write buses under different line speeds for the FL packet trace.

architecture). The main motivation behind using these traces is to see the effect of real-life packet size distributions on the architecture. The line speeds used for all the traces vary from 100 Mbps to 400 Mbps as before.

Figures 43, 44 and 45 show the variation in the utilization of the three different buses for the different line speeds. It may be noted that, as before, there is a close match between the results from the simulation and the analytical framework. Secondly, the architecture behaves almost identically for the different traces.

Recall that the bus arbitration mechanism used in our reference architecture is always first-come-first-serve (FCFS). Unfortunately, for FCFS there does not yet exist tight bounds for delay and backlog in the analytical framework that we consider here (there does exist tight bounds for static priority, round-robin, time division multiplexing, etc.). To get around this problem, we use fixed priority based arbitration mechanisms in the analytical model and compare them with FCFS used in the simulation. Towards this, one of the packet flows (recall that each trace is made up of two flows) in a trace is assigned a higher priority over the other in all the buses. For computing the end-to-end packet latency, the maximum delay experienced by the lower priority flow now gives an upper bound on the maximum delay experienced by any packet when FCFS is used. Similarly, we use the maximum delay experienced by the higher priority flow as a lower bound on the maximum delay experienced by any packet in the case of FCFS. These results are shown in Figures 46, 47 and 48 for the three different traces. Note that in all the three cases, the delay values obtained through simulation lie in between the delay values (obtained from the analytical method)

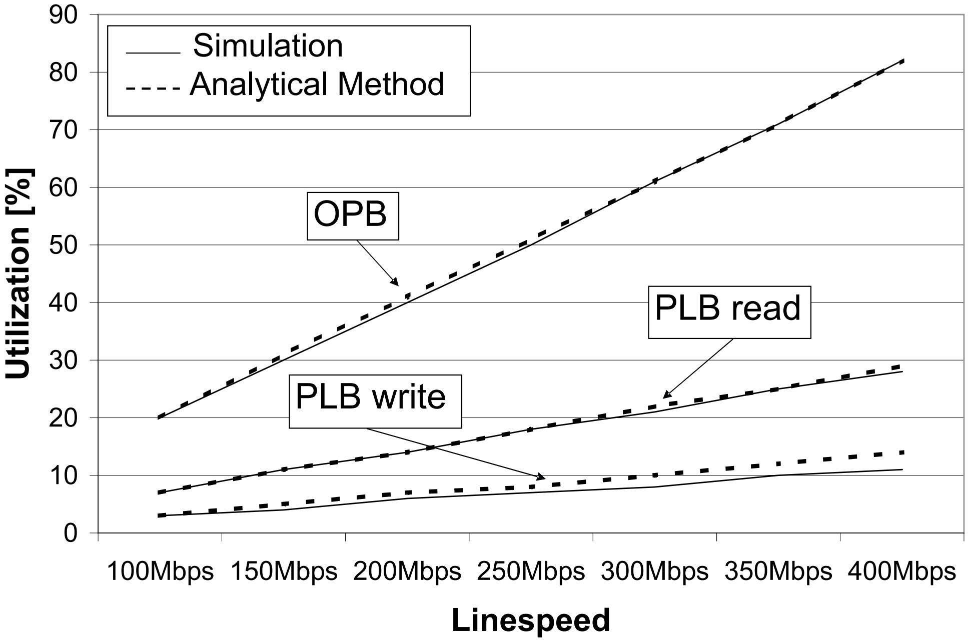

Fig. 44: The utilization of the OPB and the PLB read and write buses under different line speeds for the SDC packet trace.

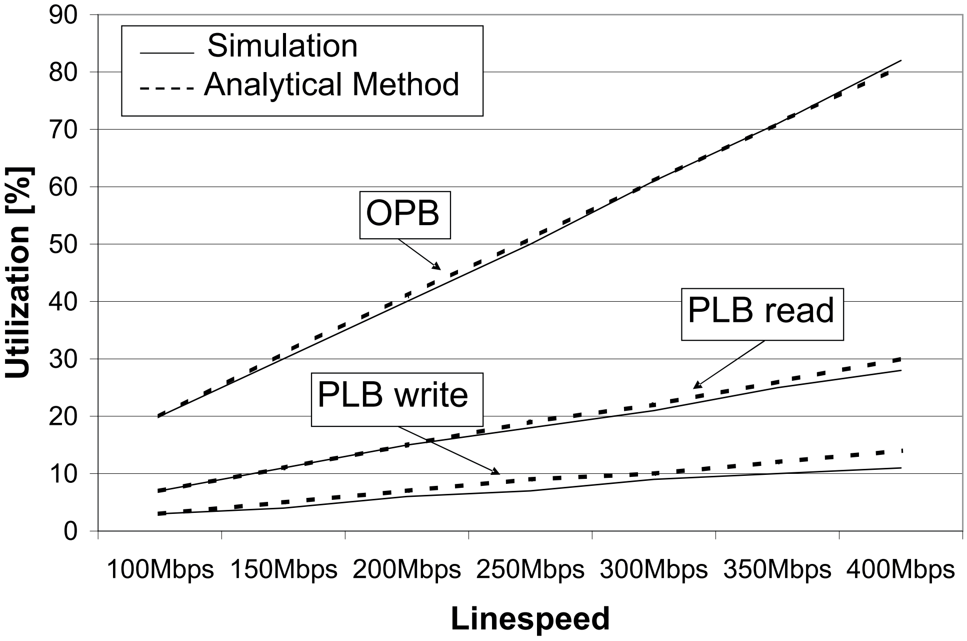

Fig. 45: The utilization of the OPB and the PLB read and write buses under different line speeds for the TAU packet trace.

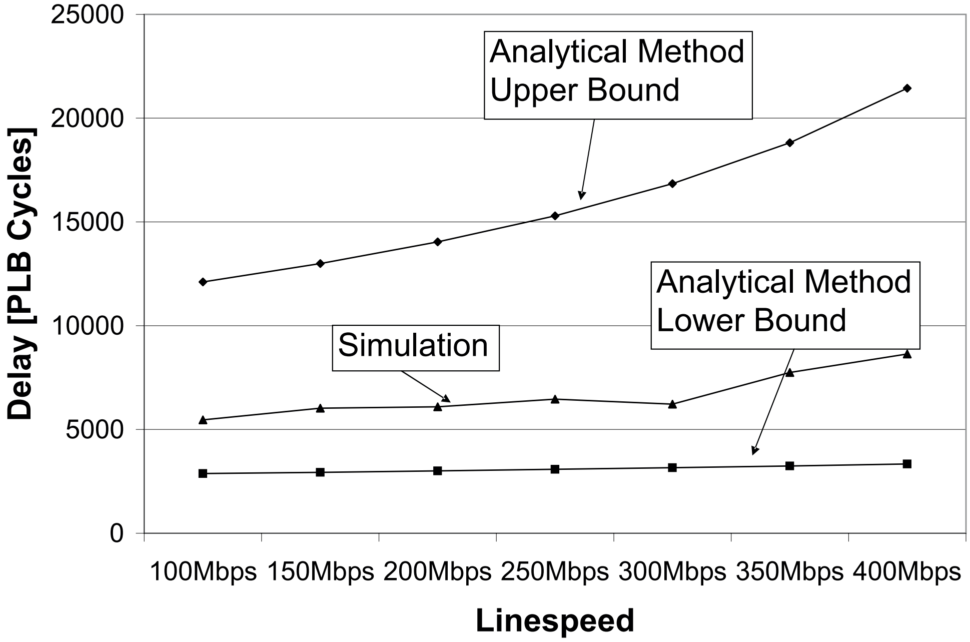

Fig. 46: The maximum end-to-end delays experienced by packets of the FL trace under different line speeds. For the two flows that make up this trace, the analytical results show the delay experienced by the higher and the lower priority flows when using priority based arbitration at the buses. The simulation results are based on FCFS implemented at all the buses.

for the high and the low priority flows. These results indicate that the architecture is sufficiently provisioned for one flow, even for high line speeds, since the maximum delays experienced by packets from the high priority remain constant with increasing line speeds for all the three buses. For the low priority flow, as expected, the delay values increase with increasing line speeds. When FCFS arbitration policy is used, the delays suffered are more than those suffered by packets from the high priority flow, but less than those suffered by the low priority flow.

In Figure 49, for each trace we first assign a high and a low priority to the two flows and measure the maximum delay experienced under this priority assignment using the analytical method. Then we reverse this priority assignment and again measure the maximum delay, and finally average the two maximum delays for each flow. Figure 49 shows this averaged maximum delay (we choose the flow for which this averaged maximum delay has a higher value) for the analytical method and the simulation results, as before, are based on FCFS at all the buses.

Lastly, Figure 50 shows the on-chip memory requirements (measured in terms of the backlog) obtained by the analytical method and by simulation. In the analytical case, as before, we use priority based arbitration at the different buses and the simulation results are based on FCFS.

It may be noted that although the analytical and the simulation results for the

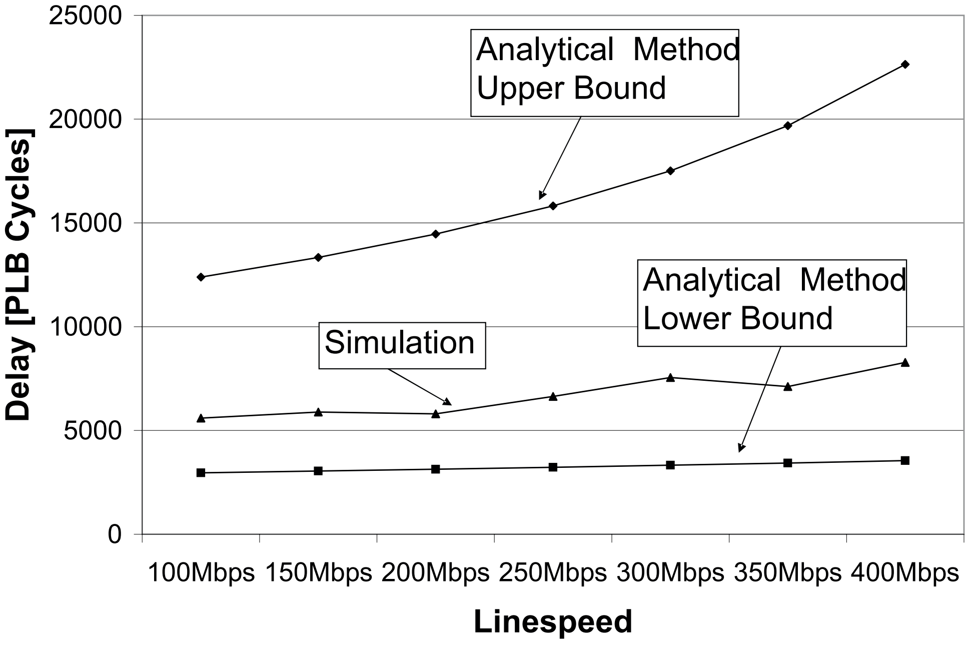

Fig. 47: The maximum end-to-end packet delays experienced by packets of the SDC trace under different line speeds.

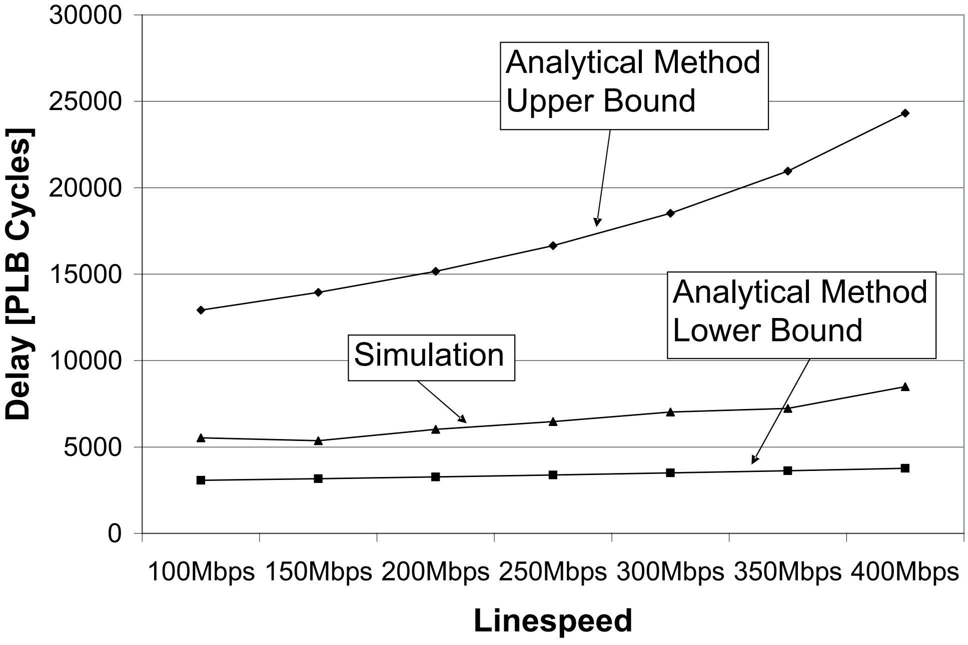

Fig. 48: The maximum end-to-end packet delays experienced by packets of the TAU trace under different line speeds.

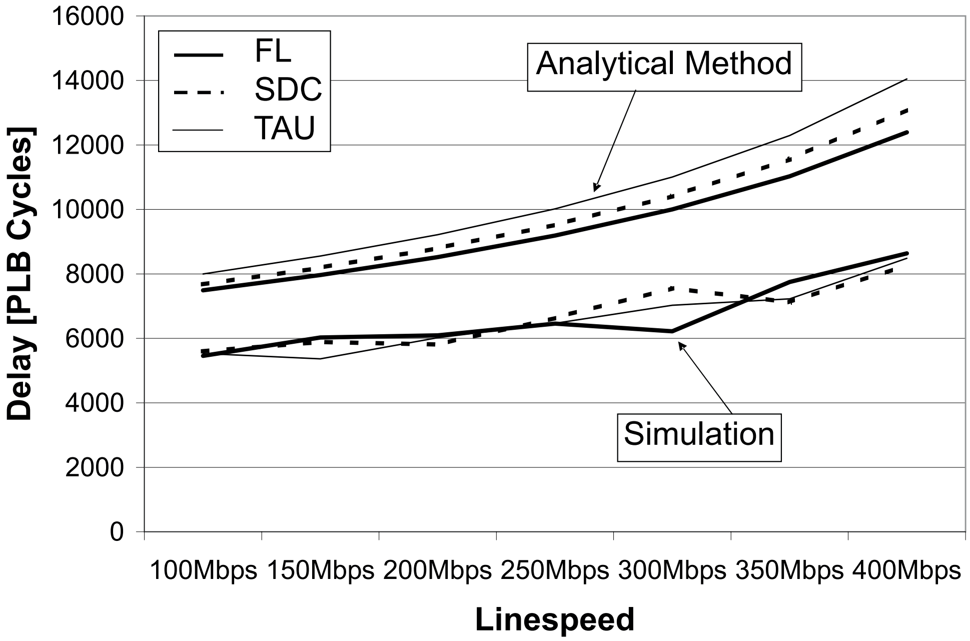

Fig. 49: The maximum end-to-end packet delays experienced by packets of all the traces under different line speeds. The plots corresponding to the analytical method shows the average of the maximum delays experienced by packets from the low and the high priority flows, when using priority based arbitration at the different buses. The simulation results are based on using FCFS at all the three buses.

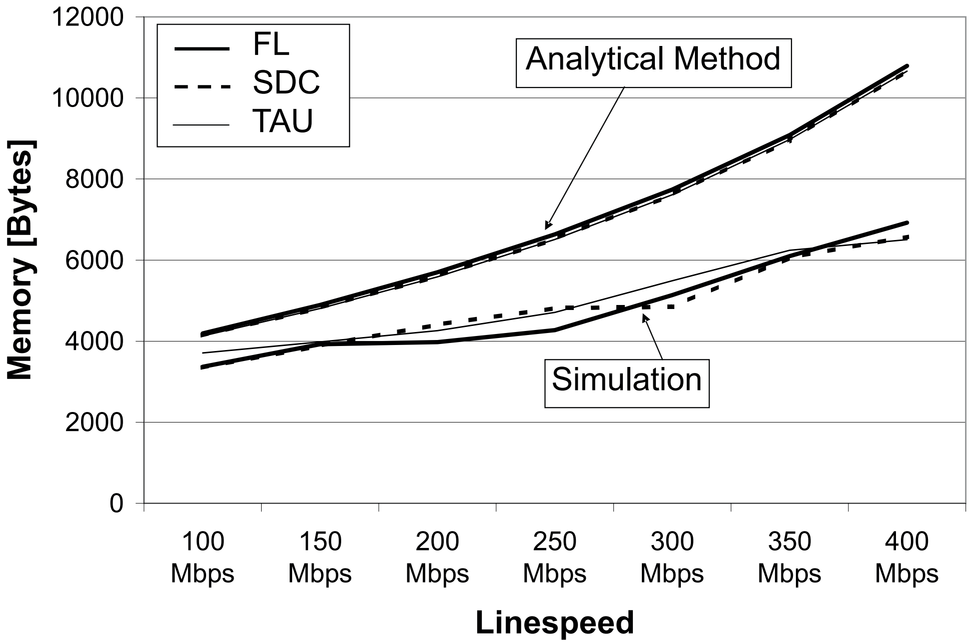

Fig. 50: The buffer memory requirements/memory fill levels for the different traces under different line speeds.

end-to-end packet delays and the memory fill levels do not match as closely as the results for the utilization values, the “trends” indicated by both the methods do match. We believe that such trends, to a large extent, would suffice to make the architectural decisions that are involved in any system-level design space exploration. One of the reasons why there is a mismatch between the values obtained by the analytical method and those obtained by simulation is that the former computes *worst case* delays and backlogs. In any particular simulation run, such worst case results may not occur. Additionally, the results obtained using the analytical framework to a very large extent depend on how tight are the different bounds for calculating the delay, backlog and resource utilization for the different scheduling policies. There is still a significant amount of work to be done in this direction (see e.g. [22]), and we hope that there will be more results in the future to improve this framework.

For all the results reported here, the simulations run in time in the order of a few minutes to several hours. In contrast to these, the analytical procedure completes execution in time less than a second for all the traces and is the only feasible option for performance evaluation in any automated design space exploration process.

4.8 The role of multiple evaluation frameworks in a design flow

4.8.1 Accuracy and evaluation times in the context of design space exploration

Traditionally, design flows for embedded processors involve architecture exploration by iterative improvement. In this approach, an initial target architecture is generated and specified in a machine description language following an analysis of the applications. Application code is then compiled for this target architecture and executed on an instruction level simulator. The performance data and statistics gathered from such a simulator is then fed into a hardware model of the target architecture to derive values of performance metrics such as power consumption, die size, etc. These values help in evaluating the architecture, identify bottlenecks and make improvements. New architectures generated through these improvements are again evaluated following the same steps and this cycle is repeated until no further improvements are possible. This whole iteration process is largely manual and ad-hoc.

However, modern embedded systems are increasingly becoming more complex and heterogeneous, and are additionally programmable. Network processors are certainly a good showcase for such trends. This makes predicting performance behavior more difficult since designing accurate simulators and performance evaluation tools for such heterogeneous systems are difficult and expensive. Further, high simulation times make automatic design space exploration impractical even during very early stages of the design.

To address these problems, new approaches to explore the design space of such systems are being considered (see for example [118]). Here, the goal is to determine an appropriate system-level architecture template during the initial phase of the design space exploration. This involves answering questions such as: Which architectural components are to be chosen? How should they be interconnected? Which tasks should be mapped to which components? Once such an architecture template is determined, further details of this template, to convert it into a detailed architecture, can then be considered. This process involves different performance evaluation techniques for the different levels of abstraction. Performance evaluation at multiple abstraction levels enable the desired speed for a design space exploration during the early stages of a design, and the accuracy required at the later stages. Modeling platforms and system-level languages such as SystemC are being developed to address these issues (see [140]).

A second possibility is to make use of fast analytical performance evaluation methods during the early stages of a design, where designers only need rough performance estimates of a template architecture and therefore a lot of architectural details can be abstracted away, making good analytical models relatively easy to construct. Once a reasonably small region of the architecture design space (captured by the template) is identified, architectures from this space can be subjected to a more extensive evaluation using cycle accurate simulation techniques. Our study in this chapter, comparing the results obtained from a system-level analytical performance evaluation method with those obtained by simulation, suggests this possibility in the context of network processors. The analytical framework considered here, to a large extent can be adapted to various abstraction levels. This depends on what each vertex in the underlying task graph described in Section 4.4 is used to model. In the experiments reported in Section 4.7, the vertices in this task graph modeled relatively low level details of an architecture such as bus communication. Nevertheless, most of the performance values obtained at this level of abstraction match fairly well with the results obtained using detailed simulation, and within a time which is orders of magnitude faster. For example, the utilization values of the different buses obtained using the analytical method match simulation results very closely. For the other performance metrics, the analytical method provides results based on which similar conclusions about the architecture can be drawn, as those from detailed simulations.

4.8.2 A design flow for packet processors

Based on the above results, one can meaningfully conclude that currently there exists analytical performance evaluation models for network processor architectures which can enable an automatic system-level design space exploration during the early to intermediate phases of a design. Since the design space at these stages can be fairly large due to the combinatorial nature of the decisions concerning the architecture that are to be made, such models can help in evalu-

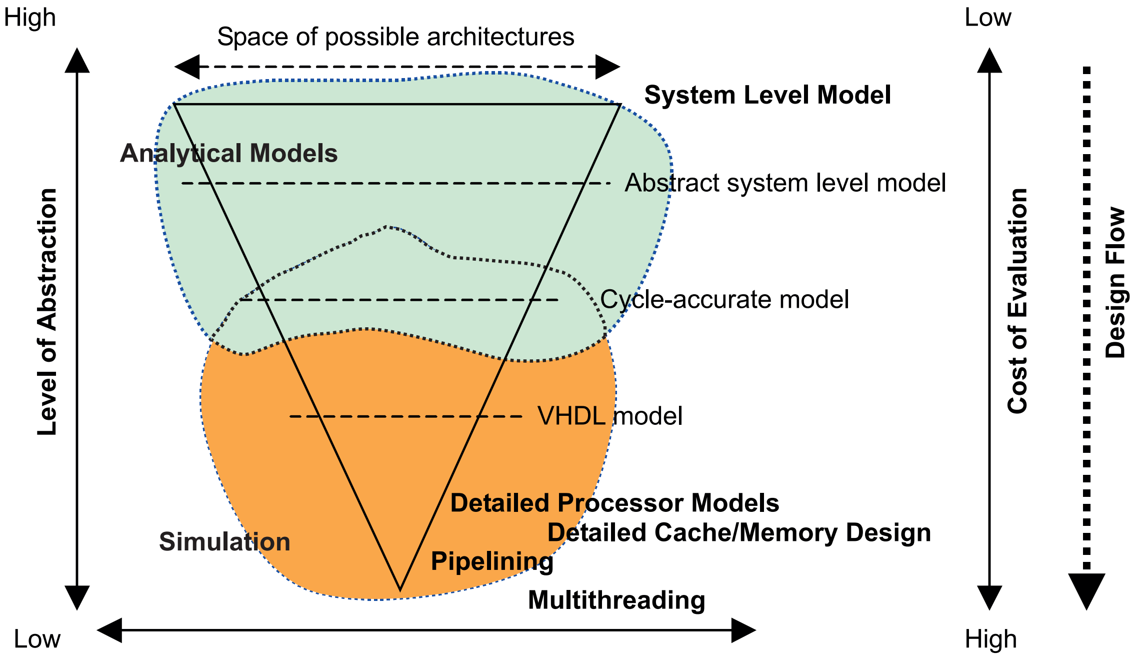

Fig. 51: Different stages of design space exploration and the associated evaluation technique.

ating a large number of designs within a short span of time. Once an interesting region of the design space is identified, in the form of one or more parameterizable template architectures, one can then resort to simulation based techniques to obtain accurate performance values of these architectures where all the lower level details are also taken into account.

This proposed design trajectory can be visualized as Figure 51. It is primarily based on the above argument that during the later stages of a design, some form of architectural template already exists—for example, it is known how many processor cores and other dedicated hardware units are to be used and how they are interconnected using the on-chip communication infrastructure, and which tasks are mapped to which processors. Design issues at this stage mostly concern the tuning of different component parameters such as bus width, cache sizes, etc., or optimally determining where various packet related data structures need to be stored. These possibilities can be exhaustively simulated to determine the optimal configuration.

4.9 Summary

We presented a detailed comparison study of an analytical performance evaluation framework for network processor architectures with a simulation based technique. The underlying assumption here has been that there is a high confidence in the simulation results. But obtaining these results is time intensive because of the high simulation times involved, thereby rendering simulation based models to be inappropriate for early staged of automated design space exploration. The main contribution of this work is a validation of the analytical model against simulation results. We believe that such cross-checking is required to establish the usefulness of such models, and also helps in identifying the appropriate design phase where such models can be used and where it is necessary to use detailed simulation to obtain meaningful results. Based on the results obtained, we proposed a design flow for network processors which relies on different classes of performance evaluation frameworks, and the two models studied here can be considered to be representatives of these classes.

We also believe that the two models considered here lie at two different extremes of a spectrum of possibilities. For evaluating different aspects of an architecture, different models might be more suitable. We envisage a suitable combination of analytical and simulation based frameworks not only across different abstraction levels of a design flow, but also within the same abstraction level, for evaluating network processor architectures. For example, it might be more suitable to use simulation to evaluate the cache/memory subsystem of an architecture, while it might be sufficient to use an analytical model to evaluate the on-chip communication architecture. The essential ingredients for such a *hybrid* framework in the context of network processors are already available. But more work needs to be done in this area to devise ways for meaningfully combining them.

5

Scheduling a mix of real-time and best-effort traffic

The problem of integrating jobs with *soft* deadlines into a hard real-time environment has been well studied in the real-time systems area (see [26] and [1] and the references therein). The motivation behind this is that multimedia tasks such as audio and video require some form of quality of service (QoS) guarantees, but their worst case execution times can vary widely. Therefore, treating them as *hard* real-time tasks would result in wasting a lot of processing resources, and underestimating their execution requirement might jeopardize other real-time tasks. Treating them as soft real-time tasks might result in some of their deadlines getting missed, but this usually does not adversely affect their quality and on the other hand improves the overall system utilization. If d is the deadline assigned to such a job and f is its finishing time, then the goal here is to minimize the mean tardiness ($\max\{0, f - d\}$) of all such jobs, without any of the hard real-time jobs missing their deadlines. Alternatively, when such jobs do not have explicit deadlines associated with them, the goal is to minimize their average response time.

All the work on this problem, however, pertains to the processor scheduling domain, and the equivalent problem in the packet scheduling domain has remained largely ignored. In the packet scheduling area, there has been an extensive amount of work on advanced buffer management and scheduling algorithms to provide QoS guarantees to real-time continuous media traffic. But relatively little has been done to exploit these algorithms to better support best-effort traffic. In the presence of a mix of real-time and best-effort traffic, the most widely followed scheme blindly gives higher priority to the real-time packets and serves the best-effort packets only if no real-time packets are present at the scheduler.

Motivated by the work done in the real-time systems area, in this chapter we attempt to address the above shortcoming by proposing a packet scheduling algorithm to provide a low delay service to best-effort packets without violating any of the delays associated with the real-time flows. Apart from improving the delays experienced by best-effort flows in general, the need for such a *low delay best-effort* service class arises in the context of serving special packets carrying, for instance, network control information such as routing table updates. Another example belonging to this class would be sporadic http requests. Since the number packets belonging to such classes are usually very small, the overhead involved in designating a distinct flow for these packets and explicitly reserving a network bandwidth to serve them is very high. As a result, such packets are treated as best-effort packets but at the same time they have a short *time-to-live*. Our proposed scheduler although continues to treat such packets as best-effort packets, would guarantee that they are delivered as early as possible without jeopardizing the deadlines associated with real-time packets.

The problem, our results and relation to previous work: The problem of integrating soft real-time or best-effort tasks into a hard real-time environment has very different manifestations and concerns in the context of processor scheduling and packet scheduling. Most of the real-time systems literature in the context of processor scheduling assume the hard real-time tasks to be periodic, with a specified period and a worst case execution requirement within each period. In several approaches based on executing the periodic tasks using a Rate Monotonic scheduling algorithm, the best-effort tasks are served using *server mechanisms* such as the Priority Exchange Server [99], the Sporadic Server [130], and the Deferrable Server [138]. All of these are based on the abstract framework of designating a special *server process*, whose *capacity* is equal to the remaining processor bandwidth after serving the periodic hard real-time tasks. This server capacity is used to service the best-effort tasks. Alternatively, slack stealing techniques under Rate Monotonic scheduling have been used in [51, 98, 146]. To better utilize system resources, dynamic priority algorithms such as the Earliest Deadline First (EDF) was used in [38, 39]. Server mechanisms under EDF were proposed in [66], where dynamic versions of the Deferrable and Sporadic Servers called the Deadline Deferrable Server and the Deadline Sporadic Server were presented. The Deadline Sporadic Server was then extended to an algorithm called the Deadline Exchange server. Lastly, five different algorithms with varying performance and complexity, for serving soft aperiodic tasks under EDF were proposed in [131, 132]. One of these algorithms, called the Total Bandwidth Server was extended to handle overload situations in [133], and an optimal algorithm for deadline assignment to soft aperiodic tasks under this scheme was proposed in [26].

All the above schemes are based on computing the remaining processor bandwidth after serving the periodic hard real-time tasks, and using this remaining bandwidth to schedule best-effort tasks. This remaining bandwidth (or utilization factor in the case of server mechanisms) is invariant over time and can therefore be specified as a single number. This is even the case when real-time tasks are not strictly periodic but their jobs are constrained by a minimum interarrival separation, or by other mechanisms different from the minimum interarrival time, such as that in the Rate-Based Execution task model [84, 85].

The setup in the case of packet scheduling is very different. Here packet arrivals from any real-time flow are constrained by *arrival curves*, which are described by general subadditive functions, each specifying the maximum amount of traffic that can arrive within any given time interval [48, 114]. For example, the (σ, ρ)-model [48] describes the worst case traffic from a flow j by a burst parameter σ_j and a long-term rate parameter ρ_j. This can be policed by a *leaky bucket mechanism* [148] and guarantees that within *any* time interval of length t, the maximum amount of traffic from flow j is bounded by $\sigma_j + \rho_j t$.

If with each such real-time flow j, a deadline d_j is associated with the interpretation that any packet from this flow has to be transmitted within d_j time units after its arrival, then it is possible to calculate the link capacity used for serving all such real-time flows. However, the *remaining link capacity* which can be used for serving best-effort flows is not invariant over time (as in the case of the remaining processor bandwidth) and can not be specified as a single number. It turns out to be a function (over time interval lengths) dependent on the arrival curves and deadlines of the real-time flows. None of the known algorithms from the processor scheduling domain extend to this case in a straightforward manner.

Deadline assignment for best-effort packets: Our algorithm is based on EDF scheduling. The real-time flows are specified using their arrival curves and a deadline is associated with each such flow. We assume that there is a single best-effort flow. This might be an aggregated flow combining multiple flows (see Section 5.5.1 for further clarifications on this). The proposed algorithm assigns a deadline to each best-effort packet (on a packet by packet basis), and schedules this packet along with the other real-time packets using EDF. Central to our algorithm is this *deadline assignment*, and we prove that the overall system always remains schedulable and the deadline assignment is optimal (in the sense that with a shorter deadline, some packet might miss its deadline).

The first algorithm we present, although optimal, is not feasible in practise since for any best-effort packet it requires the history of *all* the previous best-effort packets that arrived at the scheduler. However, based on this algorithm we propose several approximations which represent tradeoffs between the amount of computation that is required for each best-effort packet and the delay assigned to it. We show that the well known Total Bandwidth Server due to Spuri and Buttazzo [132] turn out to be one of these approximations. Our results with realistic traffic mixes consisting of audio, video, real-time transactions and best-effort flows like ftp, http and mail show between 25–45$\%$ improvements on the average in the delays experienced by the best-effort flows, without any of the real-time packets missing their deadlines.

In the next section we formally describe the model for characterizing real-time traffic in terms of arrival curves and deadlines; this forms the basis for all our algorithms. In Section 5.2 we briefly review the main concept behind the

Total Bandwidth Server and other related server mechanisms. Its connection to our algorithm is established in Section 5.4.1. We also point out the different concerns existing while designing algorithms for processor scheduling, in contrast to packet scheduling, and motivate the design issues behind our algorithm. Section 5.3 describes our optimal algorithm, following which we present our different approximation schemes in Section 5.4. Our experimental results are presented in Section 5.5.

5.1 Traffic characterization

We denote by RT the set of real-time flows with traffic arrivals to the packet scheduler, and we have one best-effort flow which might be an aggregate of several best-effort flows. Packet arrivals from the real-time flows are constrained by arrival curves [48, 114] which specify an upper bound on the amount of traffic that can arrive from a flow within any specified time interval. Packet arrivals from the best-effort flow are not constrained in anyway, and are queued up, waiting to be served.

If $a_j(t)$ denotes the traffic that arrives at the scheduler from a real-time flow j at time t, then $A_j[t, t+\tau] = \int_t^{t+\tau} a_j(t)dt$ denotes the traffic arrivals from the flow j in the time interval $[t, t+\tau]$. The maximum traffic arrival from any flow $j \in RT$ to the packet scheduler is bounded by a right-continuous subadditive *traffic constraint function* A_j^* such that for all times $t > 0$ and for all $\tau \geq 0$ we have:

$$A_j[t, t+\tau] \leq A_j^*(\tau)$$

where $A_j^*(t) = 0$ for all $t < 0$ and $A_j^*(t) \geq 0$ for $t \geq 0$.

There are usually two mechanisms to ensure that the traffic from a flow j entering the scheduler conforms to the traffic constraint function A_j^*. The first is a *traffic policer* placed at the entrance of a network node, which rejects any traffic from a flow j that does not comply to A_j^*. The second mechanism is a *rate controller* which temporarily buffers packets to ensure that the traffic from the flow j conforms to A_j^*. Example constraint functions such as that given by the (σ, ρ)-model [48] can be policed by a leaky bucket mechanism [148].

Each real-time flow j has an associated deadline d_j, and any packet from this flow must be completely transmitted within d_j time units after its arrival. We denote the deadlines assigned to (by our scheduling algorithm) the best-effort packets by δ_k, where δ_k is the (relative) deadline assigned to the k-th best-effort packet. If r_k is the arrival time of this packet then its *absolute deadline* is equal to $r_k + \delta_k$. Throughout this chapter, we refer to both absolute deadlines and relative deadlines, by the word *deadline*. It should be clear from the context which one we are referring to. Lastly, we denote the maximum packet size (from packets belonging to any flow) by s_{max}, and the transmission rate of the scheduler is equal to one.

5.2 Designing schedulers for a mix of real-time and best-effort tasks

In contrast to static priority based schemes such as those mentioned in the begining of this chapter, server mechanisms based on EDF (such as [26, 1]) achieve full processor utilization. In the Total Bandwidth Server [132] algorithm it is assumed that the processor utilization due to periodic hard real-time tasks is U_p, and the remaining capacity $U_s = 1 - U_p$ is assigned to serve best-effort tasks. Whenever a best-effort job arrives at the scheduler, it is assigned a deadline and scheduled using EDF along with the real-time jobs. If the k-th best-effort job arrives at time r_k, then it receives a deadline:

$$d_k = \max\{r_k, d_{k-1}\} + c_k/U_s$$

where c_k is the worst case execution requirement of the job. It can be shown that with this deadline assignment, the whole system remains schedulable at all points in time [132].

Note that here the schedulability of the system heavily relies on an accurate estimation of the worst case execution requirement c_k. Underestimating this might lead to some real-time job missing its deadline. Since estimating worst case execution times of jobs is considerably difficult in the context of processor scheduling, several papers have proposed means for isolating real-time jobs from best-effort jobs served in this manner using a global EDF scheduler. Well known among these is the Constant Bandwidth Server due to Abeni and Buttazzo [1]. However, note that in the context of packet scheduling, this problem does not arise since packet lengths are known upon their arrival at the link scheduler and therefore real-time packets do not run into the risk of missing their deadlines.

5.2.1 EDF versus proportional share

Our algorithm presented in this chapter is based on EDF and is similar in spirit to the Total Bandwidth Server. However, as mentioned in the begining of this chapter, the remaining link capacity available for serving best-effort packets is no longer a single number, but a function of the traffic constraint functions A_j^* (described in Section 5.1) and deadlines associated with the real-time flows. Therefore, the simple deadline calculation as done in the case of the Total Bandwidth Server does not extend to this case.

Several algorithms based on resource reservations, sometimes with dynamic feedback, have been proposed in the context of processor scheduling to support a mix of real-time and best-effort tasks [86, 111, 135, 52, 18]. Indeed, a competing alternative to our proposed scheduler, would be one based on the idea of proportional share resource allocation (such as the generalized processor sharing or any of its packetized versions) [114]. Given the arrival curves A_j^* and the deadlines d_j for each real-time flow, it is possible to deduce the minimum link bandwidth required to serve each such flow such that its deadline is satisfied, and assign the remaining bandwidth to the best-effort flow. However, schedulers based on proportional share are primarily motivated by the need for *fair sharing* of surplus link bandwidth between different flows. On the other hand, our primary goal is to greedily allocate the maximum possible link bandwidth to the best-effort flow to improve its response time, subject to the constraint that the real-time packets do not miss their deadlines. In a deterministic setup, this is the only concern since real-time packets getting served much ahead of their designated deadlines do not improve the system performance.

Our choice of EDF as a scheduling discipline is also motivated by the fact that it is known to have a larger schedulability region compared to generalized processor scheduling [64]. It has also been shown in [64] that the RSVP parameters in an IntServ framework [23] can be mapped to EDF reservations. This guarantees the conformance of our algorithm with IntServ, which happens to be one of the widely accepted service disciplines for guaranteeing real-time constraints in the Internet. Further, it was also shown in [2] that for supporting hybrid real-time multimedia applications, deadline based schemes are more appropriate compared to proportional sharing.

5.3 Optimal deadline assignment for best-effort packets

In this section we present our main algorithm. As mentioned before, it is similar to the Total Bandwidth Server of Spuri and Buttazzo [132] in the sense that best-effort packets are assigned a deadline on a packet-by-packet basis and are scheduled with the real-time packets using an EDF scheduler. However, the mechanism for deadline assignment in our case is considerably more involved and is a generalization of the algorithm for the Total Bandwidth Server.

In this section we make use of the traffic characterization defined in Section 5.1. Before describing our algorithm we need to define a few additional terms. For this, consider a mix of real-time and best-effort packets being served by the simple scheme in which real-time packets are served non-preemptively using EDF, and a best-effort packet is served only when no real-time packets are present at the scheduler. Then it can be shown using the results from [100] that the set of all real-time flows RT is schedulable if and only if for all $t \geq \min\{d_j \mid j \in RT\}$:

$$t \geq \sum_{k \in RT} A_k^*(t - d_k) + s_{max}$$

Based on this result, we define the *residual link capacity* over any time interval of length t, available to serve the best-effort flow, by a function $R_{RT}(t)$ where

$$R_{RT}(t) = t - (\sum_{k \in RT} A_k^*(t - d_k) + s_{max})$$

Recall from Section 5.1 that A_k^* is the traffic constraint function and d_k is the delay associated with the real-time flow k. s_{max} is the maximum packet length of packets belonging to any flow. We define $E_{RT}(t)$ to be the *effective residual link capacity* available within any time interval of length t to serve best-effort packets. This is given by

$$E_{RT}(t) = \min_{t' \geq t} R_{RT}(t')$$

Based on these two functions and the specifications of the real-time flows, our scheme for assigning deadlines to the best-effort packets is given by Algorithm 8.

Algorithm 8 Computing Deadlines for the Best-Effort Packets

```
Given a set $RT$ of real-time flows and one aggregated best-effort flow.
The residual link capacity $R_{RT}(t)$ to serve the best-effort flow, over any time interval of length $t$ is $R_{RT}(t) = t - (\sum_{k \in RT} A_k^*(t - d_k) + s_{max})$.
Effective residual link capacity $E_{RT}(t) = \min_{t' \geq t} R_{RT}(t')$.

**Computing the deadline $\delta_n$ of the $n$-th best-effort packet:**
  The $n$-th best-effort packet arrives at time $r_n$ and has a transmission time of $w_n$
  $\delta_n^n = \min\{t \mid E_{RT}(t) \geq w_n\}$
  **for** $i = 1$ to $n - 1$ **do**
    $\delta_n^{n-i} = \min\{t \mid E_{RT}(t) \geq \sum_{j=n-i}^{n} w_j\} - (r_n - r_{n-i})$
  **end for**
  $\delta_n = \max\{\delta_n^i \mid i = 1, 2, \ldots, n\}$
```

Theorem 15 states that with the deadline assignment given by Algorithm 8, the order in which an EDF scheduler serves them is the same as the order in which they arrive at the scheduler. Therefore, Algorithm 8 does not disrupt the order of the best-effort packets.

Thm. 15: *For any $n \geq 1$, if δ_n and δ_{n+1} are the deadlines assigned to the n-th and the $(n+1)$-th best-effort packets by Algorithm 8 then $r_{n+1} + \delta_{n+1} > r_n + \delta_n$.*

Proof: It follows from Algorithm 8 that for all $i = 0, 1, \ldots, n-1$, δ_n^{n-i} is the minimum possible value for which

$$E_{RT}(\delta_n^{n-i} + r_n - r_{n-i}) \geq \sum_{j=n-i}^{n} w_j$$

Similarly, for all $i = 0, 1, \ldots, n$, δ_{n+1}^{n+1-i} is the minimum possible value for which

$$E_{RT}(\delta_{n+1}^{n+1-i} + r_{n+1} - r_{n+1-i}) \geq \sum_{j=n+1-i}^{n+1} w_j$$

Now, for any k $(0 \leq k \leq n-1)$, consider the following two values δ_n^{n-k} and δ_{n+1}^{n-k}, defined respectively by the following two inequalities:

$$\begin{aligned} \min\{\delta_n^{n-k} \mid E_{RT}(\delta_n^{n-k} + r_n - r_{n-k}) &\geq \sum_{j=n-k}^{n} w_j\} \\ \min\{\delta_{n+1}^{n+1-(k+1)} \mid E_{RT}(\delta_{n+1}^{(n+1)-(k+1)} + r_{n+1} - r_{n+1-(k+1)}) &\geq \sum_{j=n+1-(k+1)}^{n+1} w_j\} \end{aligned}$$

Since $E_{RT}(t)$ is non-decreasing, it follows from the above that $\delta_n^{n-k} + r_n < \delta_{n+1}^{n-k} + r_{n+1}$ for $k = 0, 1, \ldots, n-1$. Hence, $\max\{\delta_n^{n-k} + r_n \mid k = 0, 1, \ldots, n-1\}$

$$\begin{aligned} &< \max\{\delta_{n+1}^{n-k} + r_{n+1} \mid k = 0, 1, \ldots, n-1\} \\ &\leq \max\{\delta_{n+1}^{n-k} + r_{n+1} \mid k = 0, 1, \ldots, n\} \end{aligned}$$

Therefore, $\delta_n + r_n < \delta_{n+1} + r_{n+1}$. □

To informally explain Algorithm 8, we introduce a family of functions $\alpha_n(t), n = 1, 2, \ldots$, where $\alpha_n(t)$ denotes the *service demand* within any time interval of length t due to a sequence of best-effort packets $1, 2, \ldots, n$, ending with the n-th packet. It means that within any time interval of length t, a sequence of best-effort packets from the set ordered $\{1, \ldots, n\}$ and ending with packet n would require a total transmission time equal to $\alpha_n(t)$ if they have to meet their assigned deadlines. Algorithm 8 assigns to each best-effort packet n the *shortest possible deadline*, maintaining the constraint that the *curve* $\alpha_n(t)$ always lies *below* the effective residual link capacity *curve* $E_{RT}(t)$. We formally prove this idea below in Theorems 16 and 17. Theorem 16 states that with the deadline assignment due to Algorithm 8, the overall system (consisting of real-time and best-effort packets) still remains schedulable, and Theorem 17 proves the optimality of the deadline assignment.

Based on the above idea of service demand functions $\alpha_n(t)$, if the first best-effort packet arrives at time r_1 and is assigned a (relative) deadline δ_1 (i.e. it has an absolute deadline equal to $r_1 + \delta_1$), then within an interval of length δ_1 the service demand due to this best-effort packet is equal to w_1. Hence $\alpha_1(\delta_1) = w_1$, and $\alpha_1(\delta) = 0$ for all $\delta < \delta_1$. Now if the second best-effort packet arrives at time r_2 and is assigned a deadline δ_2, then within an interval of length δ_2, the service demand is w_2 and within an interval of length $\delta_2 + r_2 - r_1$ the service demand is $w_1 + w_2$. These two constraints are captured in $\alpha_2(t)$, and the deadline δ_2 is chosen so that $\alpha_2(t)$ for all values of $t \geq 0$ lies below $E_{RT}(t)$. This procedure is followed for any subsequent packets (see Figure 52). Therefore, for the n-th packet, the service demands within an interval of length—δ_n is w_n, $\delta_n + r_n - r_{n-1}$ is $w_{n-1} + w_n, \ldots, \delta_n + r_n - r_1$ is $w_1 + \ldots + w_n$ (see Figure 53). As before, δ_n is chosen by Algorithm 8 such that within any of these intervals the service demands is below the effective residual link capacity available within the interval.

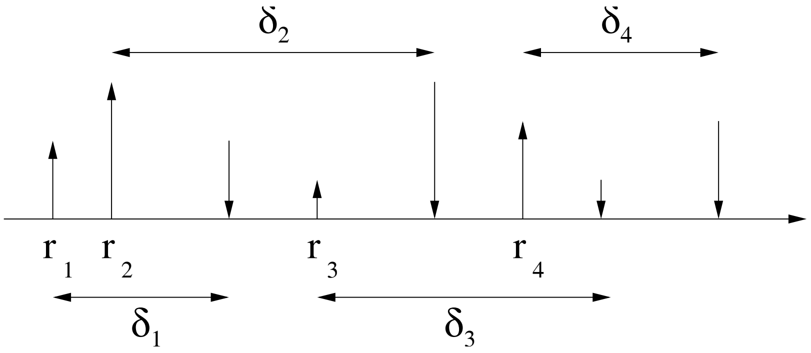

Fig. 52: A possible sequence of best-effort packet arrivals and the corresponding packet deadlines.

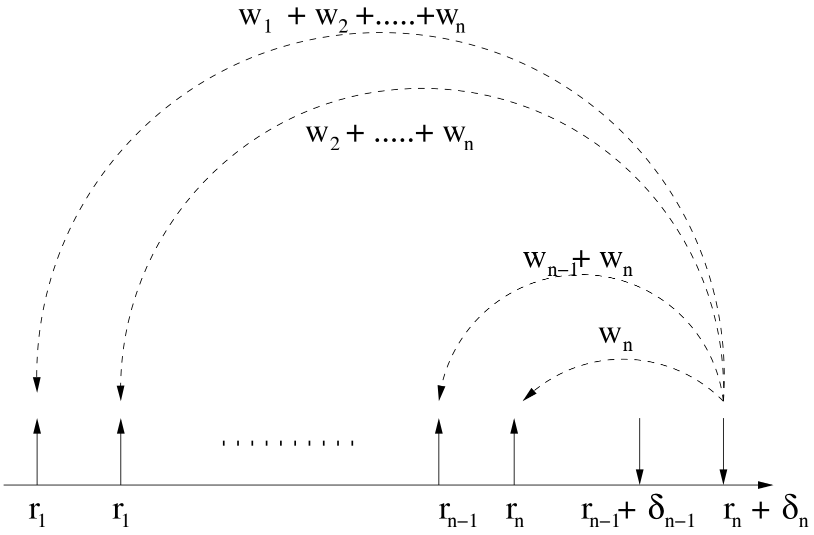

Fig. 53: Service demands arising out of the n-th best-effort packet.

5.3.1 An alternative interpretation

If we list all the above constraints for each packet, we get (see also Figure 52):

$$\begin{aligned}
\alpha_1(\delta_1) &= w_1 \\
\\
\alpha_2(\delta_2) &= w_2 \\
\alpha_2(\delta_2 + r_2 - r_1) &= \alpha_2(\delta_1 + (r_2 + \delta_2) - (r_1 + \delta_1)) = w_1 + w_2 \\
\\
\alpha_3(\delta_3) &= w_3 \\
\alpha_3(\delta_3 + r_3 - r_2) &= \alpha_3(\delta_2 + (r_3 + \delta_3) - (r_2 + \delta_2)) = w_2 + w_3 \\
\alpha_3(\delta_3 + r_3 - r_1) &= \alpha_3(\delta_1 + (r_2 + \delta_2) - (r_1 + \delta_1) + (r_3 + \delta_3) - (r_2 + \delta_2)) \\
&= w_1 + w_2 + w_3
\end{aligned}$$

$$\cdots \quad \cdots \quad \cdots$$

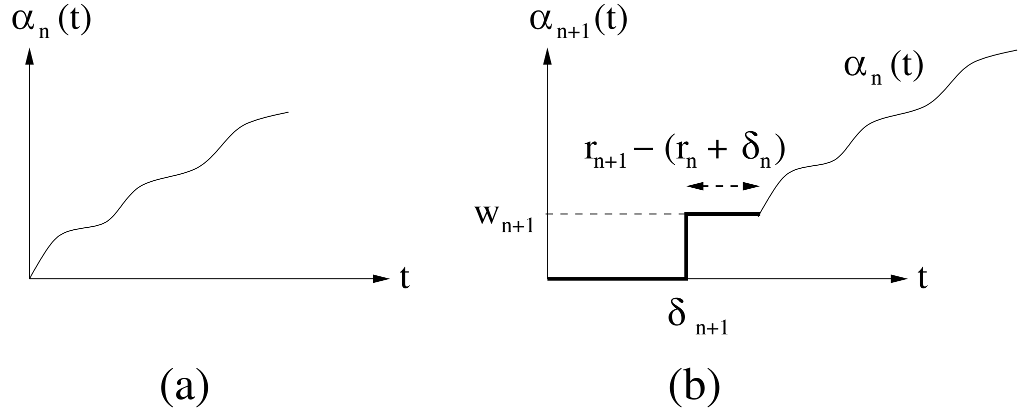

Fig. 54: Constructing the service demand curve $\alpha_{n+1}(t)$ from the curve $\alpha_n(t)$. (a) The service demand curve $\alpha_n(t)$. (b) The service demand curve $\alpha_{n+1}(t)$.

It follows from the above list of constraints that, given the curve $\alpha_n(t)$, and $r_{n+1}, \delta_{n+1}, w_{n+1}$, it is possible to construct the curve $\alpha_{n+1}(t)$ by shifting $\alpha_n(t)$ horizontally by $(r_{n+1} + \delta_{n+1}) - (r_n + \delta_n)$, vertically by w_{n+1}, and appending a step function to it as shown in Figure 54. Note that the segment (in the middle) of length $r_{n+1} - (r_n + \delta_n)$ might be positive or negative in length. In the later case, a part of the last segment of $\alpha_n(t)$ gets deleted.

With this observation, Algorithm 8 can be alternatively interpreted as follows. Given the service demand curve $\alpha_n(t)$, δ_{n+1} is the smallest possible value assigned by Algorithm 8 such that the resulting curve $\alpha_{n+1}(t)$ lies below the curve $E_{RT}(t)$. With this deadline assignment the overall system remains schedulable, and with a deadline smaller than δ_{n+1} some packet (either real-time or best-effort) might miss its deadline. This is proved next in Theorems 16 and 17.

Thm. 16: *Given a set RT of schedulable real-time flows, under the deadline assignment for best-effort packets as given by Algorithm 8, the set RT still remains schedulable and all best-effort packets are transmitted by their assigned deadlines.*

Proof: Consider a packet from a real-time flow $k \in RT$ that arrives at the scheduler at time t and is completely transmitted at time $t+\delta$. Our proof is based on bounding the workload that has to be transmitted by the scheduler before this packet is completely transmitted. For any real-time flow $j \in RT$, we denote by $A_j^{\leq x}[t, t+\tau]$ the traffic arrival from flow j during the time interval $[t, t+\tau]$ with deadlines less than or equal to x. We denote by $W^{\leq x}(y)$ the workload in the scheduler at time y due to packets with deadlines less than or equal to time x, i.e. $W^{\leq x}(y)$ is equal to the total transmission time that is required to deliver the packets waiting at the scheduler at time y, and having deadlines less than or equal to x. Lastly, let us denote by $W^{k,t}(t+\tau)$ $(0 \leq \tau \leq \delta)$ the workload in the scheduler at the time $t+\tau$, that must be served before the packet that arrived at time t from the real-time flow k can be completely transmitted (note that $W^{k,t}(t, \tau)$ includes the transmission time of this packet). Now let $t - \hat{\tau}$

($\hat{\tau} \geq 0$) be the last time before time t when the scheduler does not contain any traffic with a deadline less than or equal to the deadline of this packet, i.e. $t+d_k$. In other words, in the time interval $[t-\hat{tau}, t+\delta)$, the scheduler always contains some packet with deadline less than or equal to $t+d_k$.

$W^{k,t}(t+\tau)$ is then composed of the following terms:

- The remaining transmission time of the packet that was in transmission at time $t-\hat{\tau}$. Let us denote this by $R(t-\hat{\tau})$. By our assumption of $t-\hat{\tau}$, the deadline of this packet is greater than $t+d_k$.

- Transmission times of packets arriving in the time interval $[t-\hat{\tau}, t+\tau]$, from real-time flows with deadlines less than or equal to $t+d_k$. This is equal to $\sum_{j \in RT} A_j^{\leq t+d_k}[t-\hat{\tau}, t+\tau]$, and includes the packet from the real-time connection k that arrived at time t.

- Transmission times of best-effort packets that arrive in the time interval $[t-\hat{\tau}, t+\tau]$ and have deadlines less than or equal to $t+d_k$. Let us denote this by $BE^{\leq t+d_k}[t-\hat{\tau}, t+\tau]$.

- The length of the time interval $[t-\hat{\tau}, t+\tau]$, i.e. $\hat{\tau}+\tau$.

Since packet transmissions can not be preempted, during the time interval $[t-\hat{\tau}, t-\hat{\tau}+R(t-\hat{\tau}))$, the packet with deadline greater than $t+d_k$ is transmitted, and during the time interval $[t-\hat{\tau}+R(t-\hat{\tau}), t-\hat{\tau}]$ all the packets transmitted have a deadline less than or equal to $t+d_k$. Hence, $W^{k,t}(t+\tau)$ $(0 \leq \tau \leq \delta)$ is given as follows:

$$W^{k,t}(t+\tau) = \sum_{j \in RT} A_j^{\leq t+d_k}[t-\hat{\tau}, t+\tau] + BE^{\leq t+d_k}[t-\hat{\tau}, t+\tau] + R(t-\hat{\tau}) - (\hat{\tau}+\tau)$$

When $\tau = d_k$, the above equality becomes:

$$W^{k,t}(t+d_k) = \sum_{j \in RT} A_j^{\leq t+d_k}[t-\hat{\tau}, t+d_k] + BE^{\leq t+d_k}[t-\hat{\tau}, t+d_k] + R(t-\hat{\tau}) - (\hat{\tau}+d_k) \tag{5.1}$$

Since any packet from a real-time connection j, that arrives after the time $t+d_k-d_j$ has a deadline greater than $t+d_k$, Equation (5.1) can be written as:

$$W^{k,t}(t+d_k) = \sum_{j \in RT} A_j[t-\hat{\tau}, t+d_k-d_j] + BE^{\leq t+d_k}[t-\hat{\tau}, t+d_k] + R(t-\hat{\tau}) - (\hat{\tau}+d_k)$$

Now we would like to bound the workload $W^{k,t}(t+d_k)$. For this, we first compute an upper bound on $BE^{\leq t+d_k}[t-\hat{\tau}, t+d_k]$. Let the first best-effort packet that has an arrival time of greater than or equal to $t-\hat{\tau}$ and a deadline less than or equal to $t+d_k$, be the m-th best-effort packet (starting from the very first best-effort packet that was received by the scheduler), i.e. $r_m \geq t-\hat{\tau}$ and $r_m+\delta_m \leq t+d_k$. Similarly, let the last such best-effort packet be the n-th

($n \geq m$) packet, i.e. $r_n \geq t - \hat{\tau}$ and $r_n + \delta_n \leq t + d_k$. Then it follows from Algorithm 8 that:

$$E_{RT}(\delta_n + r_n - r_m) \geq w_m + w_{m+1} + \ldots + w_n$$

Clearly,

$$BE^{\geq t+d_k}[t - \hat{\tau}, t + d_k] = w_m + w_{m+1} + \ldots + w_n$$

Hence,

$$BE^{\geq t+d_k}[t - \hat{\tau}, t + d_k] \geq E_{RT}(\delta_n + r_n - r_m)$$

Therefore we have:

$$W^{k,t}(t+d_k) \leq \sum_{j \in RT} A_j[t-\hat{\tau}, t+d_k-d_j] + E_{RT}(\delta_n+r_n-r_m) + R(t-\hat{\tau}) - (\hat{\tau}+d_k)$$

Since $R(t - \hat{\tau}) \leq s_{max}$ and $E_{RT}(\delta_n + r_n - r_m) \leq E_{RT}(\hat{\tau} + d_k)$ (because $\delta_n + r_n - r_m \leq \hat{\tau} + d_k$), we have:

$$\begin{aligned}
W^{k,t}(t + d_k) &\leq \sum_{j \in RT} A^*_j(\hat{\tau} + d_k - d_j) + E_{RT}(\hat{\tau} + d_k) + s_{\max} - (\hat{\tau} + d_k) \\
&= \sum_{j \in RT} A^*_j(\hat{\tau} + d_k - d_j) + \min_{t' \geq (\hat{\tau}+d_k)} R_{RT}(t') + s_{max} - (\hat{\tau} + d_k) \\
&\leq \sum_{j \in RT} A^*_j(\hat{\tau} + d_k - d_j) + R_{RT}(\hat{\tau} + d_k) + s_{max} - (\hat{\tau} + d_k) \\
&= 0
\end{aligned}$$

Hence, there exists a $\tau \leq t + d_k$ such that $W^{k,t}(t + \tau) = 0$, implying that the packet from the real-time connection k, that arrived at time t, is completely transmitted on or before time $t + d_k$, thereby meeting its deadline. Using very similar arguments it is possible to show that all best-effort packets also meet their assigned deadlines. □

Thm. 17: *If any best-effort packet is assigned a deadline smaller than that assigned by Algorithm 8, then some packet (either from a real-time or from the best-effort flow) might miss its deadline.*

Proof: Let the deadline assignment for the n-th best-effort packet be less than the deadline δ_n calculated by Algorithm 8. Let this be δ'_n ($\delta'_n < \delta_n$). This implies that there exists some i, such that the sum of the transmission times of the best-effort packets $n, n-1, \ldots, i$ (i.e. $w_n+w_{n-1}+\ldots+w_i$) that have to be transmitted within an interval of length $\delta'_n+(r_n-r_i)$ exceeds $E_{RT}(\delta'_n+(r_n-r_i))$. Since for all time intervals of length $t \geq 0$, $E_{RT}(t) = \min_{t' \geq t} R_{RT}(t')$, the above implies that there exists some $j \leq i$, such that the sum of the transmission times of the best-effort packets $n, n-1, n-2, \ldots, j$ (i.e. $w_n + w_{n-1} + \ldots + w_j$) that have to be served within a time interval of length $\delta'_n + (r_n - r_j)$, so that all deadlines are met, exceeds $R_{RT}(\delta'_n + (r_n - r_j))$. Starting from the time r_j, the scheduler has to transmit the best-effort packets $j, j+1, \ldots, n$ within the next $\delta'_n + (r_n - r_j)$ time units, if all the assigned deadlines have to be met.

Assume that the scheduler is empty before the time r_j, and at time r_j^- a maximum sized packet (of size equal to s_{max}) from either a real-time flow l with $d_l > \delta'_n + (r_n - r_j)$, or from a best-effort flow with the deadline of the packet greater than $\delta'_n + (r_n - r_j)$ arrives. Starting from time r_j, packets from the real-time flows k with $d_k \leq \delta'_n + (r_n - r_j)$ arrive according to their maximum arrival rate A^*_k. Additionally, packets from the best effort flow arrive at times $r_j, r_{j+1}, \ldots, r_n$. They are assigned deadlines $\delta_j, \delta_{j+1}, \ldots, \delta_{n-1}$ according to Algorithm 8, with the exception that the n-th packet is assigned the deadline $\delta'_n < \delta_n$ as discussed above.

Since the scheduler is non-preemptive, the packet of size s_{max} that arrived at time r_j^- has to be transmitted first, before any other packet. Therefore, the total transmission time of the packets that have to be transmitted within a time interval of length $\delta'_n + (r_n - r_j)$ starting from the time r_j, if all deadlines have to be met is equal to:

$$s_{max} + \sum_{k \in RT} A_k^{\leq r_j + \delta'_n + (r_n - r_j)}[r_j, r_j + \delta'_n + (r_n - r_j)] + w_n + w_{n-1} + \ldots + w_j$$

Hence, $W^{\leq \delta'_n + (r_n - r_j)}(r_j + \delta'_n + (r_n - r_j))$

$$\begin{aligned} &> s_{max} + \sum_{k \in RT} A^*_k(\delta'_n + (r_n - r_j) - d_k) + R_{RT}(\delta'_n + (r_n - r_j)) \\ &\quad -(\delta'_n + (r_n - r_j)) \\ &= 0 \end{aligned}$$

Hence, at the time instant $r_j + \delta'_n + (r_n - r_j)$, the scheduler contains traffic with deadline less than or equal to $r_j + \delta'_n + (r_n - r_j)$, implying a deadline violation for some packet. □

It follows from the schedulability guarantee given by Theorem 16, and also the discussion in Section 5.3.1 that the deadlines assigned to the best-effort packets are dependent on their arrival rate. If too many best-effort packets arrive back-to-back and the effective residual link capacity is not too large to serve them, then the deadlines assigned to them grow larger and larger, but the real-time packets are never jeopardized. An *overload* situation created by best-effort traffic only increases their average response time. Lastly, it is clear that our scheme is never worse compared to the simple scheme of serving best-effort packets only when no real-time packets are present at the scheduler.

Note that the deadline assignment for any best-effort packet requires the entire history of all the previous packets that arrived at the scheduler. As an implementation hint (which is as well obvious to note), we point out that this history can be *reset* whenever the scheduler is idle, i.e. there are no real-time or best-effort packets with already assigned deadlines waiting to be served. The next packet arriving after such a rest can be treated as the *first* best-effort packet. For any realistic traffic flow, such resets of the scheduler will be fairly common.

5.4 Approximating the effective residual link capacity

Algorithm 8 in spite of being optimal in terms of the response time experienced by best-effort packets, is infeasible for all practical purposes. This is because it requires maintaining the history of all the best-effort packets that arrived at the scheduler, to assign a deadline to the next packet. The reason for this is that the effective residual link capacity $E_{RT}(t)$ can in general be arbitrarily shaped. Combined with the fact that packet sizes can also be arbitrary, to fit the service demand functions $\alpha_n(t)$ as tightly as possible below $E_{RT}(t)$, the full structure of $\alpha_n(t)$ is required. The complexity of the algorithm can be reduced either by approximating the effective residual link capacity $E_{RT}(t)$ by a simple function, or by approximating the packet size to allow only a fixed number of distinct sizes. In this section we consider the former approach. It should be possible to derive the approximations based on later approach from the ideas that we present in this section. For the deadline assignment for any best-effort packet, the algorithms that we present in Sections 5.4.1 and 5.4.2 require the arrival times and deadlines of the previous packet only (in contrast to *all* the previous packets). The algorithm presented in Section 5.4.3 is more involved and requires the history of a bounded number of packets, and not all. Each of these algorithms represent a tradeoff between the complexity involved in the deadline assignment and the length of the deadline assigned.

5.4.1 With a straight line passing through the origin

In this section we approximate the effective residual link capacity $E_{RT}(t)$ after serving the set of real-time flows RT, by a single straight line of slope γ passing through the origin (see Figure 55). To satisfy the schedulability condition given by Theorem 16, the slope γ is chosen to be the largest possible value such that this line lies below the exact $E_{RT}(t)$ calculated from the traffic constraint functions (or arrival curve) $A_j^*(t)$ of the real-time flows $j \in RT$, and the maximum packet size s_{max}. Therefore, $\gamma = \max\{\gamma' \mid \gamma' t \leq E_{RT}(t)\ \forall t \geq 0\}$. The deadlines of the best-effort packets are now computed with $E_{RT}^{approx}(t) = \gamma t$ as the effective residual link capacity available for transmitting the best-effort flow. We show that under this approximation, the deadline of any best-effort packet can now be optimally calculated by using parameters (such as arrival times and deadlines) belonging to the previous packet only. This is in contrast to our exact algorithm which required the arrival times of *all* the previous best-effort packets.

Consider the $(n+1)$-th best-effort packet which arrives at the time r_{n+1} and has a transmission time equal to w_{n+1}. It follows from Algorithm 8 that for the system to be schedulable the deadline assigned to this packet should satisfy:

$$\delta_{n+1} \geq w_{n+1}/\gamma \tag{5.2}$$

Let $\alpha_n(t)$ be the service demand due to the best-effort packets $1, 2, \ldots, n$ and (x, y) be the point on $\alpha_n(t)$ which is closest to $E_{RT}^{approx}(t) = \gamma t$. Then it

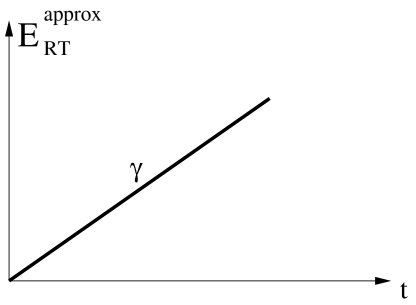

Fig. 55: Approximating $E_{RT}(t)$ with a single line passing through the origin.

follows from our discussion in Section 5.3.1 that the new coordinates of this point in $\alpha_{n+1}(t)$ become:

$$(x + r_{n+1} + \delta_{n+1} - (r_n + \delta_n), y + w_{n+1})$$

For the system to be still schedulable, we require that:

$$(x + r_{n+1} + \delta_{n+1} - (r_n + \delta_n))\gamma \geq y + w_{n+1}$$

Since (x, y) was closest to $E_{RT}^{approx}(t) = \gamma t$, if (x, y) lies below this line then all other points on $\alpha_n(t)$ are also guaranteed to lie below it.

Assuming that the system was schedulable with the deadline assignment for the n-th packet, i.e. $x\gamma \geq y$, we have:

$$(r_{n+1} + \delta_{n+1} - (r_n + \delta_n))\gamma \geq w_{n+1}$$

or,

$$\delta_{n+1} \geq w_{n+1}/\gamma + (r_n + \delta_n) - r_{n+1} \tag{5.3}$$

From inequalities (5.2) and (5.3), we get the deadline assignment for the $n + 1$-th best-effort packet to be:

$$\delta_{n+1} = w_{n+1}/\gamma + \max\{0, (r_n + \delta_n) - r_{n+1}\}$$

Therefore, $r_{n+1} + \delta_{n+1} = w_{n+1}/\gamma + \max\{r_{n+1}, r_n + \delta_n\}$ which is exactly the same as the deadline assignment in the case of the Total Bandwidth Server. This follows from the fact, that our straight line with slope γ approximating the effective residual link capacity is equivalent to a *server* with an utilization factor of γ. The Total Bandwidth Server therefore reduces to a special case of our Algorithm 8.

5.4.2 With a straight line cutting $t = \delta$

Here we approximate the effective residual link capacity $E_{RT}(t)$ again by a single straight line with slope γ, but crossing the t-axis at $t = \delta$ instead of passing

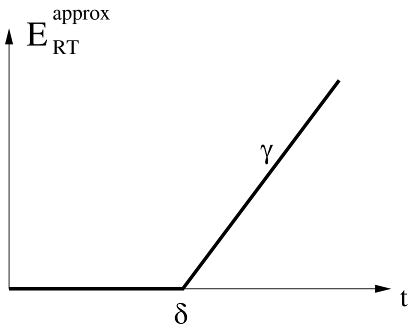

Fig. 56: Approximating $E_{RT}(t)$ with a single line cutting $t = \delta$.

through the origin as in our last approximation (see Figure 56). Therefore, it reduces to our last case if $\delta = 0$. The approximate effective residual link capacity is now given by:

$$E_{RT}^{approx}(t) = \begin{cases} 0 & \text{if } t \leq \delta \\ (t - \delta)\gamma & \text{if } t > \delta \end{cases}$$

The values γ and δ are chosen such that $E_{RT}^{approx}(t) \leq E_{RT}$ for all $t \geq 0$. Now consider the $(n+1)$-th best-effort packet that arrives at time r_{n+1} and has a transmission time of w_{n+1}. If δ_{n+1} is the deadline assigned to this packet then we require that:

$$\delta_{n+1} \geq \delta + w_{n+1}/\gamma \tag{5.4}$$

Let, as before, $\alpha_n(t)$ be the service demand due to the best-effort packets $1, 2, \ldots, n$ and (x, y) be the point on $\alpha_n(t)$ which is closest to E_{RT}^{approx}. For the new coordinate of (x, y) in $\alpha_{n+1}(t)$ to lie below E_{RT}^{approx}, after the deadline assignment δ_{n+1} to the $(n+1)$-th packet, we require:

$$(x + r_{n+1} + \delta_{n+1} - (r_n + \delta_n) - \delta)\gamma \geq y + w_{n+1}$$

Assuming that the system was schedulable with the deadline assignment of the n-th packet, i.e. $(x - \delta)\gamma \geq y$, we have:

$$(r_{n+1} + \delta_{n+1} - (r_n + \delta_n))\gamma \geq w_{n+1}$$

or,

$$\delta_{n+1} \geq w_{n+1}/\gamma + (r_n + \delta_n) - r_{n+1} \tag{5.5}$$

From inequalities (5.4) and (5.5) we have,

$$\delta_{n+1} = w_{n+1}/\gamma + \max\{\delta, (r_n + \delta_n) - r_{n+1}\}$$

5.4.3 With a combination of two line segments

To more accurately approximate the effective residual link capacity, in this section we approximate it using a combination of two line segments (instead of one

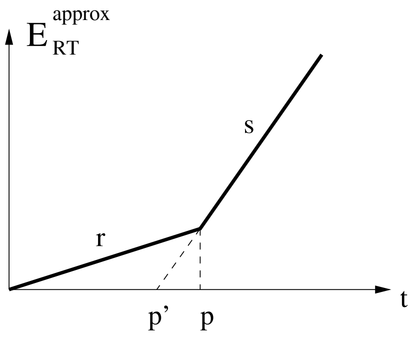

Fig. 57: Approximating $E_{RT}(t)$ with a combination of two line segments of slope r and s ($r < s$).

as in the previous cases), one of slope r passing through the origin and the second of slope s ($s > r$). The line segments intersect at a point whose t-coordinate is equal to p (see Figure 57). Hence, it is given by:

$$E_{RT}^{approx}(t) = \begin{cases} rt & \text{if } t < p \\ (t - p')s & \text{if } t \geq p \end{cases}$$

The t-intercept p' of the line with slope s can be calculated to be equal to $p - pr/s$. E_{RT}^{approx} reduces to the case in Section 5.4.1 if $r = s$, and to the case in Section 5.4.2 if $r = 0$ and $p = \delta$. As before, r, s and p are chosen such that $E_{RT}^{approx}(t) \leq E_{RT}$ for all $t \geq 0$.

Algorithm 9 gives the deadline assignment under this approximation. In contrast to the previous two algorithms, it requires the history of all the packets which constitute the portion of the service demand curve $\alpha_n(t)$ that lies to the left of $t = p$. This algorithm is based on the idea of maintaining the curve $\alpha_n(t)$ for any n only till the point $t = p$. Beyond $t = p$ only the point which is closest to the curve $E_{RT}^{approx}(t)$ is maintained. With the $(n+1)$-th packet, the deadline δ_{n+1} is chosen such that the part of $\alpha_n(t)$ that lies to the left of $t = p$ still continues to be below $E_{RT}^{approx}(t)$ in $\alpha_{n+1}(t)$, and the point on $\alpha_n(t)$ which is closest to $E_{RT}^{approx}(t)$ and lies to the right of $t = p$ also continues to lie below $E_{RT}^{approx}(t)$. Because of the horizontal shift of $\alpha_n(t)$ due to the deadline assignment to the $(n+1)$-th packet, some points on $\alpha_n(t)$ which were to the left of $t = p$ would now cross this point and one of them might become the new nearest point to $E_{RT}^{approx}(t)$ beyond $t = p$. For each packet, the algorithm therefore checks whether any new points crossing from the left to right of $t = p$ become the new nearest point to E_{RT}^{approx}.

5.4.3.1 Approximating the deadline

In the last section, the deadline assignment for any best-effort packet required the arrival and the transmission times of multiple previous packets. More precisely, for all the packets which constitute the portion of the service demand curve $\alpha_n(t)$ which was to the left of $t = p$ (i.e. $\alpha_n(t)$ for $t < p$). This was in

Algorithm 9 Computing the approximate deadline δ_{n+1} of the $(n+1)$-th best-effort packet based on approximating $E_{RT}(t)$ by a combination of two line segments

```
Given $S^n_{\leq p}$ = set of all packets $\{i \mid i \leq n$ and $(\delta_n + r_n - r_i) \leq p\}$ (i.e. the set of all packets which lie to the left of $t = p$ on the curve $\alpha_n(t)$.
Given $k_{>p}$ $(1 \leq k_{>p} \leq n)$, such that the $k_{>p}$-th packet does not belong to the set $S^n_{\leq p}$ and the point $(\delta_n + r_n - r_{k_{>p}}, \sum_{j=k_{>p}}^{n} w_j)$ on $\alpha_n(t)$ is closest to the approximate effective residual link capacity curve $E^{approx}_{RT}(t)$, among all the points on $\alpha_n(t)$ to the right of $t = p$.
The $(n+1)$-th best-effort packet arrives at time $r_{n+1}$ and has a transmission time of $w_{n+1}$.

$\{$Compute the minimum deadline to fit $\alpha_{n+1}(t)$ below $E^{approx}_{RT}(t)$, taking into account only the portion of $\alpha_n(t)$ to the left of $t = p\}$
Let $\delta_{min} = \min\{t \mid E^{approx}_{RT}(t) \geq w_{n+1}\}$
**for** all $i \in S^n_{\leq p}$ **do**
  Let $\delta = \min\{t \mid E^{approx}_{RT}(t) \geq \sum_{j=i}^{n+1} w_j\} - (r_{n+1} - r_i)$
  **if** $\delta > \delta_{min}$ **then** $\delta_{min} = \delta$ **end if**
**end for**
$\delta_{n+1} = \delta_{min}$

$\{$Compute the minimum deadline such that the point closest to $E^{approx}_{RT}(t)$ among all the points to the left of $t = p$ in $\alpha_n(t)$, continues to lie below $E^{approx}_{RT}(t)$ in $\alpha_{n+1}(t)\}$
Let $\delta = \min\{t \mid E^{approx}_{RT}(t) \geq \sum_{j=n+1}^{k_{>p}} w_j\} - (r_{n+1} - r_{k_{>p}})$
**if** $\delta > \delta_{min}$ **then** $\delta_{min} = \delta$ **end if**

$\{$Find if a new point in $\alpha_{n+1}(t)$ beyond $t = p$, becomes closer to $E^{approx}_{RT}(t)\}$
**for** all $i \in S^n_{\leq p}$ **do**
  **if** $(r_{n+1} + \delta_{n+1} - r_i) > p$ **then**
    $S^n_{\leq p} = S^n_{\leq p} \backslash \{i\}$
    **if** $(\delta_{n+1} + r_{n+1} - r_i, \sum_{j=i}^{n+1} w_j)$ is closer to the $E^{approx}_{RT}(t)$ curve compared to $(\delta_{n+1} + r_{n+1} - r_{k_{>p}}, \sum_{j=k_{>p}}^{n+1} w_j)$ **then** $k_{>p} = i$ **end if**
  **end if**
**end for**

$\{$Check if the point $(\delta_{n+1}, w_{n+1})$ on $\alpha_{n+1}(t)$ lies to the left of $t = p$, and update $S^{n+1}_{\leq p}\}$
**if** $\delta_{n+1} \leq p$ **then**
  $S^{n+1}_{\leq p} = S^n_{\leq p} \cup \{n+1\}$
**else**
  $S^{n+1}_{\leq p} = S^n_{\leq p}$
**end if**
```

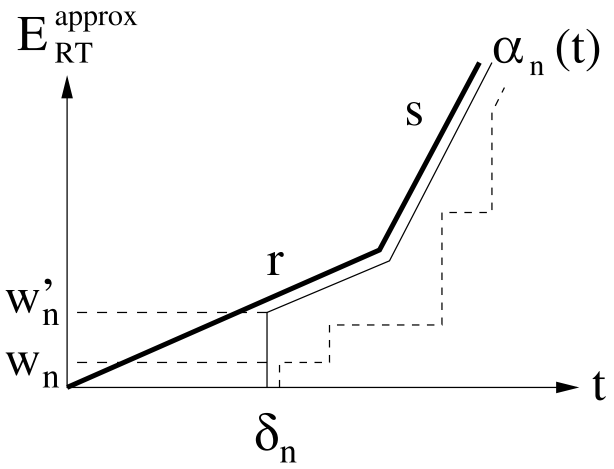

Fig. 58: Approximating the service demand curve $\alpha_n(t)$ for all n, by assuming that it coincides with $E_{RT}^{approx}(t)$. The bold line indicates the approximated effective residual link capacity, and the dashed line indicates a possible (real) service demand curve corresponding to the packet arrivals 1 to n and deadline assignments for them. The solid line in between the bold and the dashed lines indicate the approximate (or assumed) service demand curve $\alpha_n(t)$, which coincides with $E_{RT}^{approx}(t)$.

contrast to our previous two approximation schemes where the deadline calculation for any packet required the arrival time and deadline of only one previous packet. While in many cases the number of packets that constitute the portion of the service demand curve lying on the left of $t = p$ can be small (and in any case it is bounded), there might be situations where the number of such packets are very large. In the later case, computing the deadline of an incoming best-effort packet will involve an amount of computation and storage requirement which might be infeasible in practice, as in the case of our exact algorithm in Section 5.3.

In this section, apart from approximating the effective residual link capacity $E_{RT}(t)$ using a combination of two line segments, we also approximate the (optimal) deadline that can be assigned using $E_{RT}^{approx}(t)$ as the effective residual link capacity. Since we want to guarantee schedulability, the deadline assigned to any best-effort packet will now be greater than or equal to the deadline assignment with $E_{RT}^{approx}(t)$ as the effective residual link capacity, using Algorithm 9. Our approximation is based on the idea of assuming that the service demand curve $\alpha_n(t)$ at all points of time coincides exactly with the approximate effective residual link capacity $E_{RT}^{approx}(t)$. This is shown in Figure 58. Here, for the deadline assignment to the $(n+1)$-th packet, the service demand curve $\alpha_n(t)$ is assumed to be what is shown by the solid line.

If δ_{n+1} is the deadline assigned to the $n+1$-th best-effort packet, for the point (δ_{n+1}, w_{n+1}) in $\alpha_{n+1}(t)$ to lie below $E_{RT}^{approx}(t)$, the following must hold:

$$\delta_{n+1} \geq \begin{cases} w_{n+1}/r & \text{if } w_{n+1} < rp \\ (w_{n+1} - pr)/s + p & \text{if } w_{n+1} \geq rp \end{cases} \tag{5.6}$$

Algorithm 10 Approximating the deadline δ_{n+1}

```
The $(n+1)$-th best-effort packet arrives at time $r_{n+1}$ and has a transmission time of $w_{n+1}$.
The effective residual link capacity is given by $E_{RT}^{approx}(t)$.
if $\delta_n \geq p$ then
  $\{$Let $w'_n = (\delta_n - p')s$, which implies that $w'_n + w_{n+1} > pr\}$
  if $(w_{n+1} < pr)$ then $\delta_{n+1} = \max\{\frac{w_{n+1}}{r}, \frac{w_{n+1}}{s} + r_n + \delta_n - r_{n+1}\}$
  else $\delta_{n+1} = \frac{w_{n+1}}{s} + \max\{p - \frac{pr}{s}, r_n + \delta_n - r_{n+1}\}$ end if
else $\{$i.e. if $\delta_n < p\}$
  Let $w'_n = \delta_n r$
  if $w'_n + w_{n+1} \geq pr$ then
    if $(w_{n+1} < rp)$ then $\delta_{n+1} = \max\{w_{n+1}r, \frac{(w_{n+1}+\delta_n r - pr)}{s} + r_n - r_{n+1} + p\}$
    else $\delta_{n+1} = \frac{(w_{n+1}-pr)}{s} + p + \max\{0, \frac{\delta_n r}{s} + r_n - r_{n+1}\}$ end if
  else $\{$i.e. $(w'_n + w_{n+1}) < rp$, which implies that $w_{n+1} < rp\}$
    $\delta_{n+1} = \frac{w_{n+1}}{r} + \max\{0, \delta_n + r_n - r_{n+1}\}$
  end if
end if
```

If δ_n is the deadline assigned to the n-th packet and w_n is its required transmission time, then we approximate the (real) service demand curve for this packet by a service demand curve $\alpha_n(t)$ which coinsides with $E_{RT}^{approx}(t)$ (as shown in Figure 58). $\alpha_n(t)$ is given as follows:

$$\alpha_n(t) = \begin{cases} 0 & \text{if } t < \delta_n \\ w'_n & \text{if } t = \delta_n \\ E_{RT}^{approx} & \text{if } t > \delta_n \end{cases} \tag{5.7}$$

where, $w'_n = \delta_n r$ if $\delta_n < p$ and $w'_n = (\delta_n - p')s$ if $\delta_n \geq p$. In other words, Equation (5.7) is equivalent to $\alpha_n(t) = 0$ if $t < \delta_n$ and $\alpha_n(t) = E_{RT}^{approx}$ otherwise.

With δ_{n+1} as the deadline assigned to the $(n+1)$-th best effort packet, any point (x, y) on $\alpha_n(t)$ is shifted to the point (x', y') on $\alpha'_{n+1}(t)$, where $\alpha'_{n+1}(t)$ is the (real) service demand curve which is then approximated by $\alpha_{n+1}(t)$ for the deadline assignment to the $(n+2)$-th packet. (x', y') is given by,

$$(x + r_{n+1} + \delta_{n+1} - (r_n + \delta_n), y + w_{n+1})$$

Since all points on $\alpha_n(t)$ get displaced by the same amount in $\alpha'_{n+1}(t)$, the deadline δ_{n+1} should be large enough to guarantee that the point (δ_n, w_n) on $\alpha_n(t)$ when shifted to a point in $\alpha'_{n+1}(t)$, lies below $E_{RT}^{approx}(t)$. Along with Expression (5.6), this would guarantee that the entire curve $\alpha'_{n+1}(t)$ lies below $E_{RT}^{approx}(t)$. This follows from the fact that $E_{RT}^{approx}(t)$ is convex (since $s \geq r$). The deadline assignment following this scheme is given by Algorithm 10.

To understand this algorithm, note that there are two different cases to be considered when computing δ_{n+1}: (1) $\delta_n \geq p$, and therefore $w'_n = (\delta_n - p')s$, which implies that $w'_n + w_{n+1} > pr$, (2) $\delta_n < p$, and therefore $w'_n = \delta_n r$. For schedulability, in Case (1) we require that,

$$(r_{n+1} + \delta_{n+1} - r_n - p')s \geq w'_n + w_{n+1}$$

Substituting w'_n by $(\delta_n - p')s$, this is equivalent to

$$\delta_{n+1} \geq w_{n+1}/s + r_n + \delta_n - r_{n+1}$$

From Expression 5.6, it now follows that

$$\delta_{n+1} = \begin{cases} \max\{w_{n+1}/r, w_{n+1}/s + r_n + \delta_n - r_{n+1}\} & \text{if } w_{n+1} < rp \\ w_{n+1}/s + \max\{p - pr/s, r_n + \delta_n - r_{n+1}\} & \text{if } w_{n+1} \geq rp \end{cases}$$

In Case (2), the condition for schedulability depends on whether $w'_n + w_{n+1}$ is less than (Case (2a)) or greater than and equal to (Case (2b)) pr. If $w'_n + w_{n+1} < rp$, then for schedulability we require that,

$$(r_{n+1} + \delta_{n+1} - r_n)r \geq w'_n + w_{n+1}$$

Substituting w'_n by $\delta_n r$, this is equivalent to

$$\delta_{n+1} \geq w_{n+1}/r + r_n + \delta_n - r_{n+1}$$

Since in this case $w_{n+1} < rp$,

$$\delta_{n+1} = w_{n+1}/r + \max\{0, r_n + \delta_n - r_{n+1}\}$$

If $w'_n + w_{n+1} \geq rp$, for schedulability we require that,

$$(r_{n+1} + \delta_{n+1} - r_n - p')s \geq w'_n + w_{n+1}$$

Substituting w'_n by $\delta_n r$, the above inequality is equivalent to

$$\delta_{n+1} \geq w_{n+1}/s + \delta_n r/s + r_n - r_{n+1} + p'$$

Combined with the two possible cases as given by Expression 5.6, δ_{n+1}, for this subcase is given by

$$\delta_{n+1} = \begin{cases} \max\{w_{n+1}/r, (w_{n+1} + \delta_n r)/s + r_n - r_{n+1} + p'\} & \text{if } w_{n+1} < rp \\ (w_{n+1} - pr)/s + p + \max\{0, \delta_n r/s + r_n - r_{n+1}\} & \text{if } w_{n+1} \geq rp \end{cases}$$

Following the same approach, for the deadline assignment to the $(n+2)$-th packet, $\alpha'_{n+1}(t)$ (which is obtained from $\alpha_n(t)$, $\delta_(n+1)$, and w_{n+1}) is approximated by $\alpha_{n+1}(t)$, which is assumed to exactly coincide with E_{RT}^{approx}.

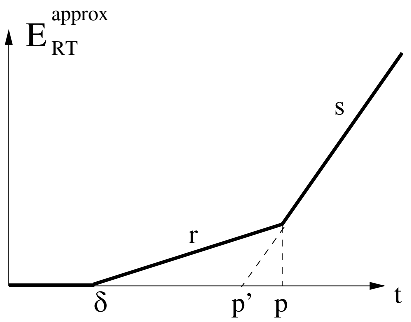

Fig. 59: Approximating $E_{RT}(t)$ with a combination of two line segments of slope r and s ($r < s$), shifted by δ.

5.4.4 With a combination of two line segments, shifted by δ

As the last case, we approximate the effective residual link capacity by a combination of two line segments, but in this case, shifted by δ. This is shown in Figure 59. The first segment has a slope equal to r, and the second has slope s, with $s > r$. The two segments meet at a point whose t-coordinate is equal to p.

The effective residual link capacity is therefore given as follows.

$$E_{RT}^{approx}(t) = \begin{cases} 0 & \text{if } t \leq \delta \\ (t-\delta)r & \text{if } \delta < t < p \\ (t-p')s & \text{if } \delta \geq p \end{cases}$$

The t-intercept of the second line segment with slope s, as shown in Figure 59, is equal to p', which can be calculated to be equal to $p - \frac{r(p-\delta)}{s}$. E_{RT}^{approx} therefore reduces to the case in Section 5.4.1 if $\delta = 0$ and $r = s$, to the case in Section 5.4.2 if $r = s$, and to the case in Section 5.4.3 if $\delta = 0$. Again, the assumption here is that δ, r, s and p are so chosen, such that $E_{RT}^{approx}(t) \leq E_{RT}(t)$ for all $t \geq 0$, and $E_{RT}^{approx}(t)$ is as close to $E_{RT}(t)$ as possible.

Algorithm 9 in the form

5.5 Experiments

In this section we describe our results from simulations modeling a 10MBit/sec access link with six different traffic classes. Three real-time (RT) flows require about two thirds of the link capacity. The single aggregated best-effort flow is obtained from three additional non real-time flows (NRT) which fairly share the residual link capacity by passing through a Weighted Fair Queuing (WFQ) based scheduler before our deadline assignment for best-effort traffic is applied. A detailed description of this mechanism along with the traffic flows, their corresponding link bandwidth requirements, and the approximations used for the deadline assignment of best-effort traffic is given below. The experimental results are presented in Section 5.5.2.

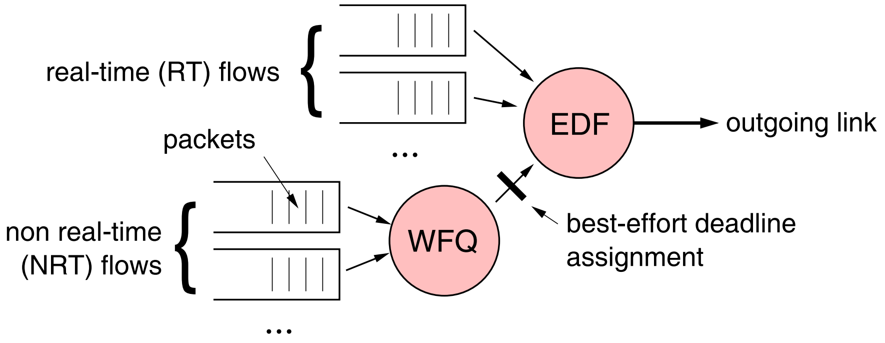

Fig. 60: Hierarchical configuration of RT and NRT schedulers.

5.5.1 Network traffic characteristics

5.5.1.1 Arrival curves and scheduling parameters

The traffic generators used for our simulations greedily generate packets as soon as they comply to a given TSpec-constrained [125] arrival curve as sketched in Figure 10 of Chapter 2. The parameters for our set of flows are given in Table 5. We distinguish three real-time and three non real-time flows. The real-time transactions class resembles traffic with a low bandwidth but hard deadline requirement to communicate with bandwidth brokers, bank applications, etc. Contrary to that, the video class models a high-bandwidth video with frame sizes considerably larger than the maximum packet length of 1536 Byte. A particular burstiness is caused by varying frame lengths due to predictive inter- or intraframe coding. Finally, the real-time voice class aggregates a couple of constant-bit-rate voice sources. The three non real-time classes can also be distinguished by varying burstiness and bandwidth requirements. For instance, the NRT mail class forms the counterpart of the RT transactions class in the set of non real-time flows since the mail class also generates moderately average rates.

Table 5 also specifies the deadlines associated with real-time flows. Our main EDF scheduler is hierarchically combined with a WFQ scheduler into which the different best-effort flows are first fed (see Figure 60). The WFQ scheduler as a result creates an ordering of the packets belonging to the different best-effort flows and results in an aggregated best-effort flow. Our deadline assignment for best-effort traffic is applied to packets belonging to this aggregated flow, after the WFQ scheduler has determined the ordering. These packets are then injected into the EDF part of the scheduler where they are scheduled along with the other real-time packets. The effective residual link capacity is therefore shared in a fair manner by scheduling the non real-time flows with the WFQ-based scheduler. The corresponding weights used by the WFQ scheduler are also stated in Table 5. The minimum packet length is bounded by 40 byte. The link capacity is set to 10 MBit/sec to model a next-generation access link from a home or an office to a service provider. Based on the parameters in Table 5 we can now derive the effective residual link capacity as shown in Fig-

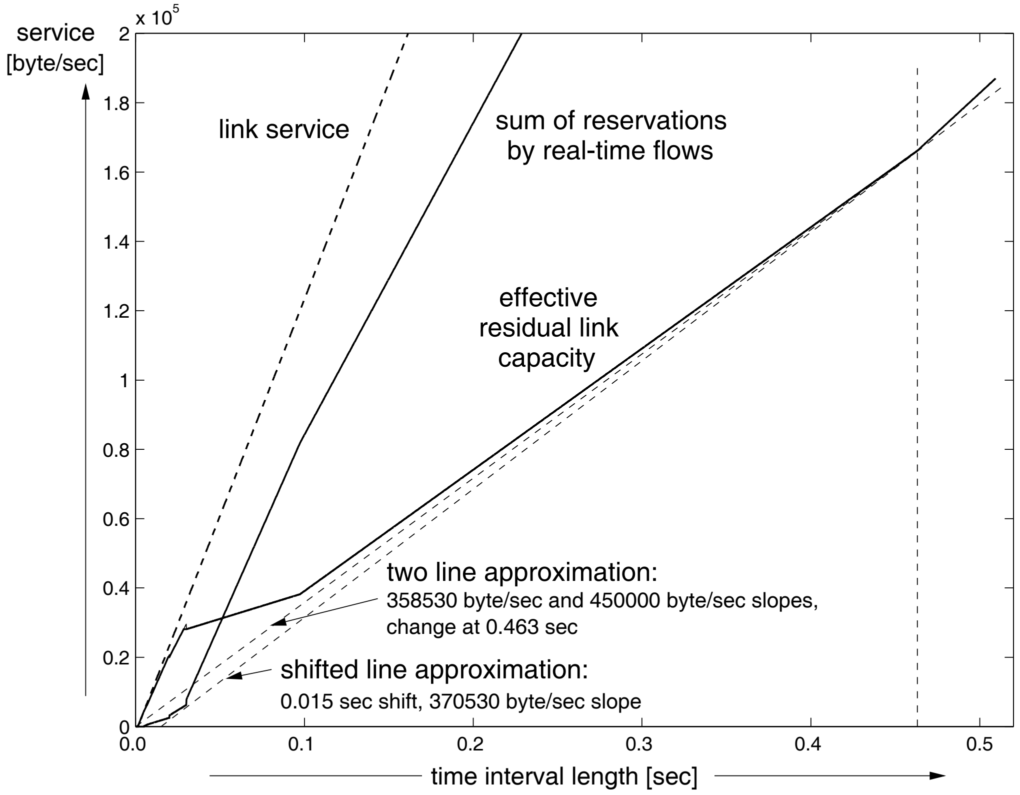

Fig. 61: Deadline assignment approximations for best-effort traffic.

ure 61. In addition, the two different approximations of the effective residual capacity used for our simulations are also sketched in this figure.

Tab. 5: TSpec description of traffic flows, see parameters in Fig. 10.

flow class	*b* [byte], *r* [byte/sec]	*M* [byte], *p* [byte/sec]	*deadline* [msec]
RT transactions	45000, 50000	700, 150000	20
RT video	15000, 600000	1536, 800000	30
RT voice	300, 150000	100, 250000	5

flow class	*b* [byte], *r* [byte/sec]	*M* [byte], *p* [byte/sec]	*WFQ weight*
NRT ftp	30720, 150000	1536, 250000	0.5
NRT http	50000, 100000	1536, 150000	0.2
NRT mail	12288, 46080	1536, 100000	0.1

5.5.1.2 Traffic generators

Since our sources generate traffic greedily according to TSpec descriptions, some more parameters are required so that we do not quickly end up in the steady section of the TSpec curve but see some burstiness at the output. We therefore consider periods when a generator is idle and not producing any packets and when the generator is enabled. We call the corresponding intervals *burst length* and *burst spacing periods* respectively. The parameters used for our simulations are given in Table 6. $N(avg, dev)$ 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. The fact that we use sources with a mean packet length above the maximum packet length leads to sequences of packets with maximum length followed by just a single or a couple of packets of smaller length. This behavior imitates the transmission of files larger than the maximum packet length which are therefore distributed over several packets. Moreover, we allow all non real-time flows to excessively use the maximum packet length to increase the negative effect on real-time flows due to non-preemptive link scheduling.

Tab. 6: Traffic generator parameters.

flow class	*packet length* [byte]	*burst length* [ms]	*burst spacing* [ms]
RT transactions	$N(300, 50)$	$U(5, 35)$	$U(50, 800)$
RT video	$N(1700, 200)$	$U(50, 100)$	$U(10, 20)$
RT voice	100	$U(2, 4)$	$U(6, 10)$
NRT ftp	$N(1700, 200)$	$U(10, 700)$	$U(100, 300)$
NRT http	$N(1700, 200)$	$U(50, 400)$	$U(50, 200)$
NRT mail	$N(1700, 200)$	$U(5, 50)$	$U(100, 300)$

5.5.2 Results

The results for the maximum and the average delay experienced by packets of the flows are given in the Tables 7 and 8. We have simulated the network node through which the traffic flows are passing, for a period of six minutes, which corresponds to about 5×10^5 processed packets. The first column shows the results for a plain best-effort (BE) scheme where best-effort traffic is transmitted only if the EDF scheduler is idle (i.e. there is no backlog from real-time flows). This column is used as a reference for a comparison with our two approximations. The second column states the results for a deadline assignment for best-effort traffic using our simple approximation of the effective residual link capacity by a shifted straight line (from Section 5.4.2). The third column shows the delay values corresponding to the more involved approximation using two line segments (from Section 5.4.3).

From the simulations it is obvious that a noticeable benefit can be gained even from the application of the very simple linear approximation of the effective residual link capacity that was presented in Section 5.4.2 to improve the

Tab. 7: Maximum delay experienced by flows within 360sec period, given in $msec$.

flow class	*plain BE scheme*	*shifted line approx.*	*two line approx.*
RT transactions	2.11 / 100%	3.36 / 159%	5.73 / 272%
RT video	3.48 / 100%	9.95 / 286%	13.97 / 401%
RT voice	1.31 / 100%	1.31 / 100%	2.54 / 194%
NRT ftp	21.95 / 100%	14.25 / 65%	7.56 / 34%
NRT http	21.29 / 100%	16.14 / 76%	11.14 / 52%
NRT mail	28.82 / 100%	22.84 / 79%	16.69 / 58%

Tab. 8: Average delay experienced by flows within 360sec period, given in $msec$.

flow class	*plain BE scheme*	*shifted line approx.*	*two line approx.*
RT transactions	0.77 / 100%	0.86 / 117%	1.00 / 130%
RT video	1.52 / 100%	1.83 / 120%	1.88 / 124%
RT voice	0.53 / 100%	0.53 / 100%	0.62 / 117%
NRT ftp	2.50 / 100%	1.87 / 75%	1.73 / 69%
NRT http	2.61 / 100%	1.93 / 74%	1.78 / 68%
NRT mail	3.61 / 100%	2.28 / 63%	1.97 / 55%

response time for best-effort traffic. If we are able to afford the more involved approximation using two segments, then further improvement in the delays experienced by the best-effort traffic can be achieved. From the results, it is also possible to recognize that the real-time traffic with the most loose deadline (RT video) experiences the largest slow down. Of course, in spite of the injection of the best-effort traffic with deadlines into the EDF scheduler, none of the deadlines associated with the real-time traffic are missed. This is because we only exploit the inherent mobility (in time) of the real-time packets due to the unused link capacity. The representations of the simulation trace in Figures 62 and 63 underpins the positive effect of our algorithm on the responsiveness for the non real-time flows. For readability reasons, only two selected flows are displayed.

Note that the maximum delay experienced by some of the best-effort traffic improves by almost 65%, whereas the average delay improves by almost 45% in some cases. In the case of best-effort flows like sporadic http requests such benefits can be immediately perceived, proving the effectiveness of our scheduler.

5.6 Summary

In this chapter we presented a packet scheduling algorithm to improve the response time of best-effort packets in the presence of real-time traffic flows. Although this algorithm is infeasible to implement in practise, we presented sev-

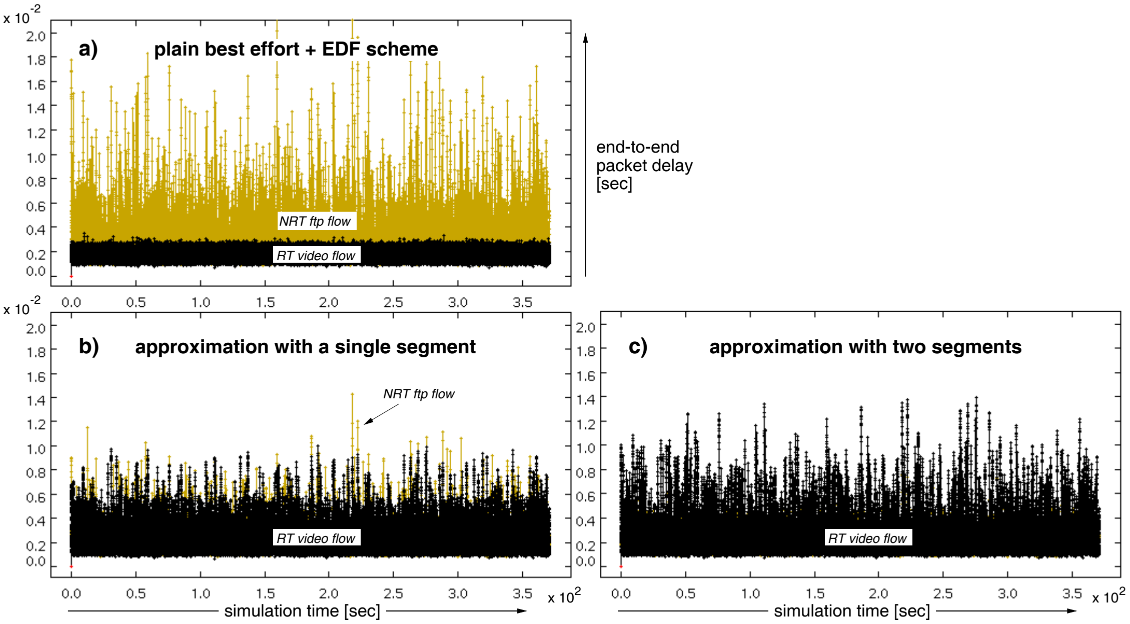

Fig. 62: Comparison of the delay experienced by two (out of the six) selected flows using our different approximation schemes for best-effort service. (a) The delays experienced by a RT and a NRT flow when NRT packets are served only when no RT packets are present at the scheduler. (b) The delays experienced when our approximation scheme in Section 5.4.2 is used. (c) Further improvement in the delay experienced by the NRT flow using our approximation scheme in Section 5.4.3.

eral approximations based on this optimal algorithm, which represent tradeoffs between complexity and performance. All of our algorithms exploit the mobility in time of real-time packets with deadlines relatively far in the future to inject best-effort packets into the scheduler without violating any real-time deadlines. Thus, we reduce the delay experienced by non real-time flows since we do not require the EDF scheduler to be idle before best-effort traffic is eligible for service. Our experiments used a combination of WFQ and EDF schedulers. WFQ was used to fairly distribute the remaining link bandwidth among the different best-effort flows, and EDF was used as an overall scheduling strategy to benefit from its larger schedulability region compared to WFQ.

We are aware of improved EDF-based scheduling disciplines such as service curve-based EDF (SCED [123]) and would like to point here that our deadline assignment for best-effort traffic can also be applied to this case by calculating the effective residual link capacity based on the schedulability test of SCED (eq. (14) in [123]). In the same way it is possible to refine the definition of fair shares for best-effort flows by using the Fair Service Curve approach in [110].

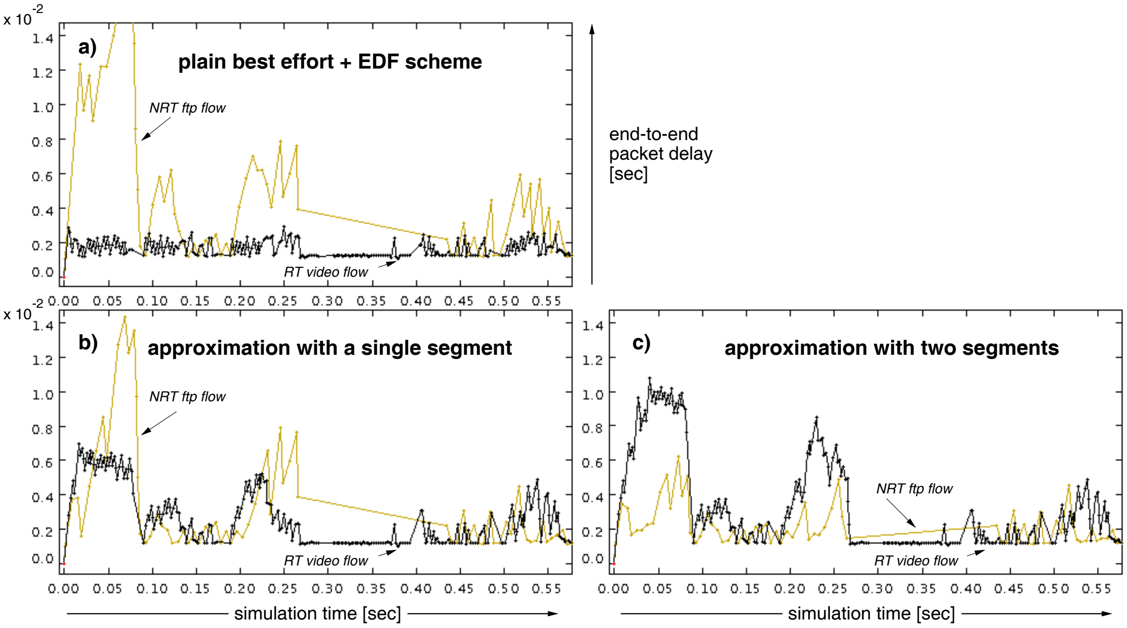

Fig. 63: Comparison of the delay experienced by the two selected flows shown in Figure 62 Here an excerpt from the simulation trace given in Figure 62 is shown.

6

Concluding remarks

In this thesis we looked into several issues related to timing analysis and scheduling for embedded packet processing devices. These devices have emerged only very recently, to manage the growing complexity of packet processing tasks and to support high line speeds. Both these aspects—timing analysis and scheduling—have already been widely studied in the context of system-level design of traditional embedded systems. However, these studies have largely focussed on data-dominated applications like digital signal processing and the developed techniques do not directly hold for control-dominated applications like packet processing. We have identified the main differences between these two classes, and motivated by these differences we proposed a model for tasks and architectures, especially suited to capture the characteristics of one particular class of control-dominated applications: network packet processing.

Based on these models, we then proposed several algorithms for analysing the hardware and software architectures of packet processors. In this regard, the main results that we have obtained can be summarized as follows.

- We posed the problem of determining the feasibility of a mapping of the different packet processing tasks onto the different architectural components of a packet processor, as a schedulability analysis problem. This was motivated by the fact that any feasible mapping should (i) be able to process packets at line speed, and (ii) the QoS requirements associated with the different real-time packet flows (such as those arising from voice or video processing applications) show be satisfied.

  Our underlying model for the schdulability analysis problem was based on a very recently proposed real-time task model called the *recurring real-time task model* [15]. However, known the schedulability analysis problem for this model had exponential complexity and no hardness results for this problem was known. We showed that the schedulability analysis problem in our model is NP-hard, but can be solved in pseudo-polynomial time. These results also carry over to the recurring real-time task model and answer several questions that were raised in [15].

Further, we introduced a novel concept called *approximate schedulability analysis*, using which the schedulability problem can be solved in polynomial time if a small error in the decisions made by the algorithm is allowed. Using this concept, we were able to demonstrate that in spite of the intractability of schedulability analysis for the proposed model, it can nevertheless be solved in reasonable (polynomial) time for all practical purposes. We also showed that this concept is not only restricted to the proposed model, but is applicable to a wide variety of other real-time task models for which only exponential or pseudo-polynomial time algorithms were known till now.

One of the most important "plus-points" of our theoretical results is that, in contrast to many approximation algorithms developed in the context optimization algorithms [79], our algorithms are implementable and lead to clear improvements in running time by several factors when compared to the exponential or pseudo-polynomial time algorithms.

- For system-level timing analysis, we have considered the theory of *real-time calculus*, which was first proposed by Thiele *et al.* in [145], and used for analysing packet processing architectures in [142, 143]. Using this theory, along with a model of tasks, architectures and packet flows, it is possible to answer questions about system-level timing, which in turn lead to insights about different system properties of a packet processor, such as the on-chip and off-chip memory requirements and the load generated on the different processing and communication resources.

  We extended this work in two ways. Firstly, we showed that the results derived with the framework composed of the task, resource, and flow models, and the real-time calculus, generalizes many results from the real-time systems area. Secondly, using detailed experiments we showed that the results from this framework also compare well with detailed cycle-accurate simulations. Based on this, we then proposed a methodology for the design space exploration of packet processing architectures to take into account the large design space involved.

- Traffic management is one of the main functions of any packet processor, especially in the case of routers. Often, the requirement is that the link scheduler show be so designed, that the packets from all real-time flows are sent out in a timely fashion, but at the same time, the best-effort flows also maintain a reasonable throughput. In this context, we proposed a novel scheduler which takes into account the time constraints associated with packets belonging to real-time flows, but at the same time provides the "best-possible service" to non-real-time or best effort flows. We formalized this concept of "best-possible service" and showed that in the context of the proposed scheduler, there can be theoretical guarantees gives on this. It turned out that the theory behind our scheduler generalizes a number of service schemes developed in the real-time systems area for integrating soft-real-time jobs into a hard-real-time environment, in the context of processor scheduling.

The experimental results obtained by implementing the proposed scheduler on a realistic mix of different classes of traffic showed that it leads to significant improvements in the service received by best-effort packets, when compared to the usual policy of simply assigning all best-effort traffic classes the lowest priority. In the context of access networks, where there can be flows comprised of sporadic http requests, the improvements achieved by our scheduler are clearly perceivable.

All of the above results apart from being interesting in the context of network packet processors, also integrate and generalize models and algorithms existing in the areas of real-time systms, networking, and system-level design.

6.1 Future work

Our work also gives rise to several new questions, and there also exist possibilities for improving some of the results we have presented. A few of these are listed below.

- The algorithms for schedulability analysis that we presented did not make any special assumptions about the task graphs—they were considered to be general directed acyclic graphs. Although, be believe that this is not an over-generalization, since task graphs arising in the context of packet processing applications can be arbitrarily complex, it might nevertheless be interesting to see if more efficient algorithms can be designed for restricted classes of task graphs. One reasonable possibility would be to assume that the different possible branches are annotated with probabilities, each indicating the probability that the branch is taken. Such probabilities can be derived from profiling the code corresponding to the application being analysed. The question here would then be to find if the complexity of the schedulability analysis can be improved, if we can tolerate some vertices missing their deadlines with low probability. This is in contrast to the nature of the error our algorithms currently make—some vertices can miss their deadlines, but not by a large amount of time.

  There might also be other possibilities for improvement. For example, the (pseudo-polynomial) upper bound on the maximum number of tests to check if the sum of the demand-bound functions for any time interval of length t, exceed t, is a worst-case bound. For any given problem instance, the number of changes in the sum of the demand-bound function can actually be much smaller. We showed this in the context of a restricted task model, where the recurring behaviour of the task graphs were not taken into account. But it would be interesting to investigate this possibility and exploit it in the general setup.

- There are several open issues to be investigated in the context of the framework for timing analysis that we considered in Chapter 4. Firstly, the bounds that are currently known for specifying the output characteristics of a packet stream in terms of its input characteristics, are not tight for many scheduling disciplines—policies like FCFS can not be reasonably modeled because the known bounds are too loose.

  Secondly, in the current setup, the characteristics of an output packet stream in terms of its input characteristics is determined from the scheduling discipline and the other packet streams that are also being processed by the resource. It would be interesting to see if a more more general system-theoretic approach can be developed, where the processor characteristics are specified as a function which transform an input packet stream into an output stream.

  In the context of the framework for design space exploration presented in the same chapter, there are several possibilities for extension. Firstly, in the case of any heterogeneous architecture, not all architectural components might be amenable to analysis using the analytical method. This is especially true for caches, since the performance in this case is dependent on data locality which is difficult to model in our framework. It would therefore be natural to have a performance evaluation framework where some architectural components are evaluated using simulation based approaches, while others are evaluated using analytical models. In connection to this, it should also be investigated how a design space exploration methodology should incorporate such a hybrid evaluation framework.

  Secondly, it might be noted that the framework considers abstract models of network traffic—each arrival curve captures an entire class of trafic flows, all of which have certain properties which are same and are captured by the parameters in the arrival curve. This is in contrast to simulation based approaches where only traffic traces are used. Based on the performance of an architecture on one trace, no theoretical guarantees about the architecture can be given in terms of an entire class of traces. Following this same approach, it might be investigated if it is possible to formulate such an abstraction for the architectures as well, and then perform a *symbolic design space exploration*.

- In the context of the integrated scheduling of real-time and best-effort traffic, one possible direction for further work would be to put the algorithms in a more general framework. Presently, the top level scheduler is based on EDF, and the basic principle behind the algorithm is to exploit the deadline-slack for the real-time packets and insert best-effort packets by postponing the transmission of real-time packets.

  In [58], a framework was presented for general deadline-ordered service disciplines (EDF is one particular case of a deadline-ordered service discipline). Using this framework, it might be possible generalize our proposed algorithm, such that it would be possible to compute the slack in the case of any deadline-ordered scheduling policy and then use this slack to serve best-effort packets.

Bibliography

[1] L. Abeni and G.C. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. 19th IEEE Real-Time Systems Symposium*, pages 4–13. IEEE Computer Society Press, 1998.

[2] L. Abeni, G. Lipari, and G.C. Buttazzo. Constant bandwidth vs proportional share resource allocation. In *Proc. IEEE International Conference on Multimedia Computing and Systems*, volume 2, pages 107–111, 1999.

[3] S.G. Abraham, B.R. Rau, and R. Shreiber. Fast design space exploration through validity and quality filtering of subsystem designs. Technical Report HPL-2001-220, Compiler and Architecture Research Program, Hewlett Packard Laboratories, August 2000.

[4] A. Agarwal. Performance tradeoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[5] J. Allen, B. Bass, C. Basso, R. Boivie, J. Calvignac, G. Davis, L. Freléchoux, M. Heddes, A. Herkersdorf, A. Kind, J. Logan, M. Peyravian, M. Rinaldi, R. Sabhikhi, M. Siegel, and M. Waldvogel. PowerNP network processor: Hardware, software and applications. *IBM J. Res. & Dev.*, 2003 (*to appear*).

[6] *Alpha Architecture Reference Manual, Digital Press*, 1992.

[7] AMBA specification overview, ARM.
`http://www.arm.com/pro+peripherals/amba/`.

[8] F. Arts, P. Barri, I. Clemminck, A. Niemegeers, B. Pauwels, G. Taildeman, and M. Vrana. Network processor requirements and benchmarking. *Computer Networks*, 41(5):549–562, April 2003.

[9] ATM Forum Technical Committee. ATM user-network interface (UNI) specification version 3.1, September 1994.

[10] S. Audenaert and P. Chandra (NPF Benchmarking Working Group co-chairs. Network processors benchmark framework. `http://www.npforum.org/`.

[11] N.C. Audsley, K.W. Tindell, and A. Burns. The end of the line for static cyclic scheduling? In *Proc. Euromicro Conf. on Real-Time Systems*, Finland, 1993. IEEE Computer Society Press.

[12] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Publishers, 1997.

[13] S. Baruah. Feasibility analysis of recurring branching tasks. In *Proc. 10th Euromicro Workshop on Real-Time Systems*, pages 138–145, Berlin, Germany, 1998.

[14] S. Baruah. Personal communication, Department of Computer Science, University of North Carolina at Chapel Hill, February 2002.

[15] S. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1):93–128, January 2003.

[16] S. Baruah, D. Chen, S. Gorinsky, and A.K. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999.

[17] S. Baruah, A.K. Mok, and L.E. Rosier. The preemptive scheduling of sporadic, real-time tasks on one processor. In *Proc. 11th IEEE Real-Time Systems Symposium (RTSS)*, pages 182–190, Orlando, Florida, 1990. IEEE Computer Society Press.

[18] A. Bavier and L.L. Peterson. BERT: A scheduler for best effort and real-time tasks. Technical Report TR-602-99, Department of Computer Science, Princeton University, 2001.

[19] J.C.R. Bennett and H. Zhang. WF^{2}Q: worst-case fair weighted fair queueing. In *Proc. IEEE INFOCOM*, volume 1, pages 120–128, 1996.

[20] J.C.R. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, October 1997.

[21] R.A. Bergamaschi, S. Bhattacharya, R. Wagner, C. Fellenz, M. Muhlada, W.R. Lee, F. White, and J-M. Daveau. Automating the design of SoCs using cores. *IEEE Design & Test of Computers*, 18(5):32–45, 2001.

[22] J.-Y. Le Boudec and P. Thiran. *Network Calculus - A Theory of Deterministic Queuing Systems for the Internet*. Lecture Notes in Computer Science 2050, Springer Verlag, 2001.

[23] B. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet architecture: an overview. Request for Comments 1633, Internet Engineering Task Force (IETF), June 1994.

[24] J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Intl. Journal of Computer Simulation*, 4:155–182, 1994.

[25] D.C. Burger and T.M. Austin. The SimpleScalar tool set. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, 1997.

[26] G.C. Buttazzo and F. Sensini. Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments. *IEEE Transactions on Computers*, 48(10):1035–1052, October 1999.

[27] W. Bux, W.E. Denzel, T. Engbersen, A. Herkersdorf, and R.P. Luijten. Technologies and building blocks for fast packet forwarding. *IEEE Communications Magazine*, 39(1):70–77, January 2001.

[28] S. Chakraborty, T. Erlebach, S. Künzli, and L. Thiele. Schedulability of event-driven code blocks in real-time embedded systems. In *Proc. 39th Design Automation Conference (DAC)*, pages 616–621, New Orleans, LA, June 2002. ACM Press.

[29] S. Chakraborty, T. Erlebach, and L. Thiele. On the complexity of scheduling conditional real-time code. In *Proc. 7th International Workshop on Algorithms and Data Structures (WADS)*, Lecture Notes in Computer Science 2125, pages 38–49. Springer Verlag, 2001.

[30] S. Chakraborty, M. Gries, and L. Thiele. Supporting a low delay best-effort class in the presence of real-time traffic. In *Proc. 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54, San Jose, California, September 2002. IEEE Press.

[31] S. Chakraborty, S. Künzli, and L. Thiele. Approximate schedulability analysis. In *Proc. 23rd IEEE International Real-Time Systems Symposium (RTSS)*, pages 159–168, Austin, Texas, December 2002. IEEE Press.

[32] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proc. 6th Design, Automation and Test in Europe (DATE)*, Munich, Germany, March 2003.

[33] S. Chakraborty, S. Künzli, L. Thiele, A. Herkersdorf, and P. Sagmeister. Performance evaluation of network processor architectures: Combining simulation with analytical estimation. *Computer Networks*, 41(5):641–665, April 2003.

[34] P.R. Chandra, F. Hady, R. Yavatkar, T. Bock, M. Cabot, and P. Mathew. Benchmarking network processors. In Crowley et al. [47], chapter 2, pages 11–25. A preliminary version of this paper appeared in the Proc.

1st Workshop on Network Processors, held in conjunction with the 8th International Symposium on High-Performance Computer Architecture, Cambridge, Massachusetts, 2002.

[35] C.S. Chang. Stability, queue length and delay, Part 1: Deterministic queuing networks. Technical Report RC 17708, IBM, 1992.

[36] C.S. Chang. *Performance Guarantees in Communication Networks*. Springer-Verlag, 2000.

[37] H.J. Chao and J.S. Hong. Design of an ATM shaping multiplexer with guaranteed output burstiness. *Int. J. Comput. Syst. Sci. & Eng., (Special issue on ATM Switching)*, 12(2):131–141, 1997.

[38] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, 1989.

[39] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2:181–194, 1990.

[40] K.-H. Cho and H. Yoon. Design and analysis of a fair scheduling algorithm for QoS guarantees in high-speed packet-switched networks. In *Proc. IEEE International Conference on Communications (ICC)*, volume 3, pages 1520–1525, 1998.

[41] Y.-P. Chu and E-H. Hwang. A new packet scheduling algorithm: minimum starting-tag fair queueing. *IEICE Transactions on Communications*, E80-B(10):1529–1536, October 1997.

[42] N. Cravotta. Network processors: The sky is the limit. *EDN-Magazine, US-edition*, 44(24):108–119, November 1999.

[43] P. Crowley and J-L. Baer. A modeling framework for network processor systems. In Crowley et al. [47], chapter 8, pages 167–188. A preliminary version of this paper appeared in the Proc. 1st Workshop on Network Processors, held in conjunction with the 8th International Symposium on High-Performance Computer Architecture, Cambridge, Massachusetts, 2002.

[44] P. Crowley, M. Fiuczynski, and J.-L. Baer. On the performance of multithreaded architectures for network processors. Technical Report 2000-10-01, Department of Computer Science, University of Washington, 2000.

[45] P. Crowley, M.E. Fiuczynski, J-L. Baer, and B.N. Bershad. Workloads for programmable network interfaces. In *Proc. 2nd Annual IEEE Workshop on Workload Characterization*, Austin, TX, October 1999. Also appears

as Chapter 7, in *Workload Characterization for Computer System Design*, Kluwer Academic Publishers, 2000.

[46] P. Crowley, M.E. Fiuczynski, J-L. Baer, and B.N. Bershad. Characterizing processor architectures for programmable network interfaces. In *Proc. International Conference on Supercomputing*, pages 54–65, Santa Fe, 2000.

[47] P. Crowley, M.A. Franklin, H. Hadimioglu, and P.Z. Onufryk, editors. *Network Processor Design: Issues and Practices, Volume 1*. Morgan Kaufmann Publishers, San Francisco, CA, 2003.

[48] R.L. Cruz. A calculus for network delay, Part I: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.

[49] R.L. Cruz. A calculus for network delay, Part II: Network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, 1991.

[50] J.A. Darringer, R.A. Bergamaschi, S. Bhattacharya, D. Brand, A. Herkersdorf, J.K. Morrell, I.I. Nair, P. Sagmeister, and Y. Shin. Early analysis tools for system-on-a-chip design. *IBM J. Res. & Dev.*, 46(6):691–707, November 2002.

[51] R.I. Davis, K.W. Tindell, and A. Burns. Scheduling slack time in fixed priority preemptive systems. In *Proc. Real-Time Systems Symposium*, pages 222–231, 1993.

[52] D. de Niz and R. Rajkumar. Chocolate: A reservation-based real-time java environment on windows/nt. In *Proc. 6th IEEE Real Time Technology and Applications Symposium*, 2000.

[53] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *Internetworking: Research and Experience*, 1(1):3–26, September 1990.

[54] G. Dittmann. Personal communication, IBM Zürich Research Laboratory, Switzerland, November 2002.

[55] G. Dittmann and A. Herkersdorf. Network processor load balancing for high-speed links. In *Proc. International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, pages 727–735, San Diego, California, July 2002.

[56] G. Dittmann and A. Herkersdorf. Design methodology for control-dominated ASIPs, 2003. *Unpublished manuscript.*

[57] R. Ernst. Codesign of embedded systems: Status and trends. *IEEE Design & Test of Computers*, pages 45–54, April 1998.

[58] N.R. Figueira and J. Pasquale. A schedulability condition for deadline-ordered service disciplines. *IEEE/ACM Transactions on Networking*, 5(2):232–244, 1997.

[59] M.A. Franklin and T. Wolf. A network processor performance and design model with benchmark parameterization. In Crowley et al. [47], chapter 6, pages 117–140. A preliminary version of this paper appeared in the Proc. 1st Workshop on Network Processors, held in conjunction with the 8th International Symposium on High-Performance Computer Architecture, Cambridge, Massachusetts, 2002.

[60] M.A. Franklin and T. Wolf. Power considerations in network processor design. In *Proc. 2nd Workshop on Network Processors, held in conjunction with the 9th International Symposium on High-Performance Computer Architecture*, USA, March 2003.

[61] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, Englewood Cliffs, N.J., 1994.

[62] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[63] L. Georgiadis, R. Guérin, and A.K. Parekh. Optimal multiplexing on a single link: delay and buffer requirements. *IEEE Transactions on Information Theory*, 43(5):1518–1535, September 1997.

[64] L. Georgiadis, R. Guérin, V. 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.

[65] L. Geppert. The new chips on the block [network processors]. *IEEE Spectrum*, 38(1):66–68, January 2001.

[66] T.M. Ghazalie and T.P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9(1):31–67, 1995.

[67] T. Givargis, F. Vahid, and J. Henkel. System-level exploration for pareto-optimal configurations in parameterized system-on-a-chip. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, San Jose, 2001. Also to appear in the IEEE Transactions on Very Large Scale Integration Systems.

[68] S.J. Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proc. IEEE INFOCOM*, volume 2, pages 636–646, June 1994.

[69] P. Goyal, H.M. Vin, and H. Cheng. Start-time fair queuing: a scheduling algorithm for integrated services packet switching networks. *Computer Communication Review*, 26(4):157–168, October 1996.

[70] M. Gries. *Algorithm-Architecture Trade-offs in Network Processor Design*. PhD thesis, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology (ETH) Zürich, 2001.

[71] M. Gries, C. Kulkarni, C. Sauer, and K. Keutzer. Comparing analytical modeling with simulation for network processors: A case study. In *Proc. of the Designer's Forum at the 6th Design, Automation and Test in Europe (DATE)*, Munich, Germany, March 2003.

[72] M. Gries, C. Kulkarni, C. Sauer, and K. Keutzer. Exploring trade-offs in performance and programmability of processing element topologies for network processors. In *Proc. 2nd Workshop on Network Processors, held in conjunction with the 9th International Symposium on High-Performance Computer Architecture*, USA, March 2003.

[73] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, Boston, May 2002.

[74] R. Guérin, S. Kamat, V. Peris, and R. Rajan. Scalable QoS provision through buffer management. *Computer Communication Review*, 28(4):29–40, October 1998.

[75] R. Haas, C. Jeffries, L. Kencel, A. Kind, B. Metzler, R. Pletka, M. Waldvogel, L. Freléchoux, and P. Droz. Creating advanced functions on network processors: Experience and perspectives. manuscript, November 2002.

[76] E.L. Hahne and R.G. Gallager. Round robin scheduling for fair flow control in data communication networks. In *Proc. IEEE International Conference on Communications*, volume 1, pages 103–107. IEEE, New York, NY, USA, 1986.

[77] J. Heinanen and R. Guérin. A two rate three color marker. Request for Comments 2698, Internet Engineering Task Force (IETF), September 1999.

[78] D. Herity. Network processor programming. *Embedded Systems Programming*, 14(8):33–52, 2001.

[79] D.S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, Boston, MA, 1997.

[80] Blue Logic technology, IBM.
`http://www.chips.ibm.com/bluelogic/`.

[81] Coreconnect bus architecture, IBM. http://www.chips.ibm.com/products/coreconnect/.

[82] J.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 (ETH) Zürich, Switzerland, June 2000.

[83] J.W. Janneck and M. Naedele. Modeling hierarchical and recursive structures using parametric petri nets. In *Proc. High Performance Computing (HPC)*, pages 445–452, San Diego, 1999.

[84] K. Jeffay and D. Bennett. A rate-based execution abstraction for multimedia computing. In *Proc. 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, LNCS 1018, pages 64–75, 1995.

[85] K. Jeffay and S. Goddard. Rate-based resource allocation models for embedded systems. In *Proc. 1st Workshop on Embedded Software (EMSOFT)*, LNCS 2211, pages 204–222. Springer-Verlag, 2001.

[86] M.B. Jones, D. Rosu, and M-C. Rosu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proc. 16th ACM Symposium on Operating System Principles*, pages 198–211, 1997.

[87] C.R. Kalmanek, H. Kanakia, and S. Keshav. Rate controlled servers for very high-speed networks. In *Proc. GLOBECOM*, volume 1, pages 12–20. IEEE, New York, NY, USA, 1990.

[88] S. Karlin. *Embedded Computational Elements in Extensible Routers*. PhD thesis, Department of Computer Science, Princeton University, January 2003.

[89] S. Karlin and L. Peterson. VERA: An extensible router architecture. *Computer Networks*, 38(3):277–293, February 2002.

[90] V. Kathail, S. Aditya, R. Schreiber, B.R. Rau, D.C. Cronquist, and M. Sivaraman. PICO: Automatically designing custom computers. *IEEE Computer*, 35(9):39–47, September 2002.

[91] L. Kencel and J-Y. Le Boudec. Adaptive load sharing for network processors. In *Proc. IEEE INFOCOM*, June 2002.

[92] K. Keutzer, S. Malik, R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonolization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design*, 19(12), 2000.

[93] E. Kohler. *The Click Modular Router*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2000.

[94] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[95] D.C. Ku and G. De Micheli. *High-level Synthesis of ASICs under Timing and Synchronization Constraints*. Kluwer Academic Publishers, Boston, MA, 1992.

[96] K. Lahiri, A. Raghunathan, and S. Dey. System level performance analysis for designing on-chip communication architectures. *IEEE Trans. on Computer Aided-Design of Integrated Circuits and Systems*, 20(6):768–783, 2001.

[97] E.A. Lee. Overview of the ptolemy project. Technical Memorandum UCB/ERL M01/11, March 2001. University of California, Berkeley.

[98] J.P. Lehoczsky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proc. Real-Time Systems Symposium*, pages 110–123, 1992.

[99] J.P. Lehoczsky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proc. Real-Time Systems Symposium*, pages 261–270, 1987.

[100] J. Liebeherr, D.E. Wrege, and D. Ferrari. Exact admission control for networks with a bounded delay service. *IEEE/ACM Transactions on Networking*, 4(6):885–901, 1996.

[101] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[102] National Laboratory for Applied Network Research (NLANR), Traces collected in June 2000, `http://pma.nlanr.net/pma/`.

[103] H. De Man, I. Bolsens, B. Lin, K. van Rompaey, S. Vercauteren, and D. Verkest. Co-design of DSP systems. In G. De Micheli and M. Sami, editors, *Hardware/Software Co-Design*, pages 75–104. Kluwer Academic Publishers, 1996.

[104] G. Memik, W. Mangione-Smith, and W. Hu. NetBench: A benchmarking suite for network processors. In *Proc. Intl. Conference on Computer Aided Design (ICCAD)*, pages 39–42, 2001.

[105] A.K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, MIT, 1983. Available as Technical Report No. MIT/LCS/TR-297.

[106] A.K. Mok and D. Chen. A multiframe model for real-time tasks. In *Proc. 17th Real-Time Systems Symposium (RTSS)*, Washington, D.C., 1996. IEEE Computer Society Press.

[107] A.K. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering*, 23(10):635–645, 1997.

[108] The Moses project: Modeling, Simulation, and Evaluation of Systems, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH) Zürich, Switzerland. `http://www.tik.ee.ethz.ch/~moses`.

[109] J.B. Nagle. On packet switches with infinite storage. *IEEE Transactions on Communications*, 35(4):435–438, April 1987.

[110] T. S. E. Ng, D. Stephens, I. Stoica, and H. Zhang. Supporting best-effort traffic with Fair Service Curve. In *GLOBECOM'99*, pages 1799–1807, December 1999.

[111] J. Nieh and M.S. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proc. 16th ACM Symposium on Operating System Principles*, pages 184–197, 1997.

[112] A. Österling, T. Benner, R. Ernst, D. Herrmann, T. Scholz, and W. Ye. *Hardware/Software Co-Design: Principles and Practice*, chapter The COSYMA System. Kluwer Academic Publishers, 1997.

[113] A.K. Parekh. *A Generalized processor sharing approach to flow control in integrated services networks*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1992. Number LIDS-TH-2089.

[114] A.K. Parekh and R.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.

[115] A.K. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The multiple node case. *IEEE/ACM Transactions on Networking*, 2(2):137–150, April 1994.

[116] P.G. Paulin, C. Pilkington, and E. Bensoudane. StepNP: A system-level exploration platform for network processors. *IEEE Design & Test of Computers*, pages 17–26, November–December 2002.

[117] M. Peyravian and J. Calvignac. Fundamental architectural considerations for network processors. *Computer Networks*, 41(5):587–600, April 2003.

[118] A.D. Pimentel, P. Lieverse, P. van der Wolf, L.O. Hertzberger, and E.F. Deprettere. Exploring embedded-systems architectures with Artemis. *IEEE Computer*, 34(11):57–63, November 2001.

[119] IBM PowerNP NPe405 Embedded Processors. `http://www-3.ibm.com/chips/techlib/techlib.nsf/products/PowerNP_NPe405_Embedded_Processors`.

[120] B.R. Rau and M.S. Schlansker. Embedded computer architecture and automation. *IEEE Computer*, 34(4):75–83, April 2001.

[121] K. Richter and R. Ernst. Model interfaces for heterogeneous system analysis. In *Proc. 6th Design, Automation and Test in Europe (DATE)*, Munich, Germany, March 2002.

[122] K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. Model composition for scheduling analysis in platform design. In *Proc. 39th Design Automation Conference (DAC)*, New Orleans, LA, June 2002. ACM Press.

[123] H. Sariowan, R.L. Cruz, and G.C. Polyzos. SCED: A generalized scheduling policy for guaranteeing Quality-of-Service. *IEEE/ACM Transactions on Networking*, 7(5):669–684, October 1999.

[124] Seamless Hardware/Software Co-Verification, Mentor Graphics. `http://www.mentor.com/seamless/`.

[125] S. Shenker and J. Wroclawski. General characterization parameters for integrated service network elements. Request for Comments 1633, Internet Engineering Task Force (IETF), September 1997.

[126] H. Shimonishi and T. Murase. A network processor architecture for flexible QoS control in very high speed line interfaces. In *Proc. IEEE Workshop on High Performance Switching and Routing*, pages 402–406, May 2001.

[127] M. Shreedhar and G. Varghese. Efficient fair queuing using Deficit Round-Robin. *IEEE/ACM Transactions on Networking*, 4(3):375–385, June 1996.

[128] G. Snider. Spacewalker: Automated design space exploration for embedded computer systems. Technical Report HPL-2001-220, Compiler and Architecture Research Program, Hewlett Packard Laboratories, September 2001.

[129] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a robust software-based router using network processors. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, pages 216–229, Banff, Alberta, Canada, October 2001.

[130] B. Sprunt, L. Sha, and J.P. Lehoczsky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1:27–60, 1989.

[131] M. Spuri and G.C. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proc. IEEE Real-Time Systems Symposium*, 1994.

[132] M. Spuri and G.C. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2):179–210, 1996.

[133] M. Spuri, G.C. Buttazzo, and F. Sensini. Robust aperiodic scheduling under dynamic priority systems. In *Proc. 16th IEEE Real-Time Systems Symposium*, pages 210–221, 1995.

[134] J.A. Stankovic, M. Spuri, K. Ramamritham, and G.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.

[135] D.C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proc. 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 145–158, 1999.

[136] D. Stiliadis and A. Varma. Efficient fair queueing algorithms for packet-switched networks. *IEEE/ACM Transactions on Networking*, 6(2):175–185, April 1998.

[137] D. Stiliadis and A. Varma. Rate-proportional servers: a design methodology for fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 6(2):164–174, April 1998.

[138] J.K. Strosnider, J.P. Lehoczsky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, 1995.

[139] B. Suter, T.V. Lakshman, D. Stiliadis, and A.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.

[140] SystemC homepage. `http://www.systemc.org`.

[141] H. Takada and K. Sakamura. Schedulability of generalized multiframe task sets under static priority assignment. In *Proc. 4th International Workshop on Real-Time Computing Systems and Applications (RTCSA)*, pages 80–86, 1997.

[142] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. A framework for evaluating design tradeoffs in packet processing architectures. In *Proc. 39th Design Automation Conference (DAC)*, pages 880–885, New Orleans, LA, June 2002. ACM Press.

[143] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. Design space exploration of network processor architectures. In Crowley et al. [47], chapter 4, pages 55–90. A preliminary version of this paper appeared in the Proc. 1st Workshop on Network Processors, held in conjunction with the 8th International Symposium on High-Performance Computer Architecture, Cambridge, Massachusetts, 2002.

[144] L. Thiele, S. Chakraborty, M. Gries, A. Maxiaguine, and J. Greutert. Embedded software in network processors – models and algorithms. In *Proc. 1st Workshop on Embedded Software (EMSOFT)*, Lecture Notes in Computer Science 2211, pages 416–434, Lake Tahoe, CA, USA, 2001. Springer Verlag.

[145] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104, 2000.

[146] T-S. Tia, J.W-S. Liu, and M. Shankar. Algorithms and optimality of scheduling soft aperiodic requests in fixed-priority preemptive systems. *Real-Time Systems*, 10(1):23–43, 1996.

[147] M. Tsai, C. Kulkarni, N. Shah, K. Keutzer, and C. Sauer. A benchmarking methodology for network processors. In Crowley et al. [47], chapter 7, pages 141–165. A preliminary version of this paper appeared in the Proc. 1st Workshop on Network Processors, held in conjunction with the 8th International Symposium on High-Performance Computer Architecture, Cambridge, Massachusetts, 2002.

[148] J.S. Turner. New directions in communications (or which way to the information age?). *IEEE Communications Magazine*, 25(8):8–15, 1986.

[149] The Cadence Virtual Component Co-design (VCC). `http://www.cadence.com/products/vcc.html`.

[150] M. Venkatachalam, P. Chandra, and R. Yavatkar. A highly flexible, distributed multiprocessor architecture for network processing. *Computer Networks*, 41(5):563–586, April 2003.

[151] T. Wolf. *Design and Performance of a Scalable High-Performance Programmable Router*. PhD thesis, Department of Computer Science, Washington University in St. Louis, May 2002.

[152] T. Wolf and M. Franklin. CommBench - A telecommunications benchmark for network processors. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 154–162, Austin, Texas, 2000.

[153] T. Wolf, M.A. Franklin, and E.W. Spitznagel. Design tradeoffs for embedded network processors. Technical Report WUCS-00-24, Department of Computer Science, Washington University in St. Louis, 2000.

[154] T. Wolf, P. Pappu, and M. Franklin. Predictive scheduling of network processors. *Computer Networks*, 41(5):601–621, April 2003.

[155] F. Worm. A performance evaluation of memory organizations in the context of core based network processor designs. Master's thesis, Institut Eurécom, Sophia-Antipolis, France, This work was done at IBM Research Laboratory Zürich, 2001.

[156] H. Zhang and D. Ferrari. Rate-controlled static-priority queuing. In *Proc. IEEE INFOCOM*, volume 1, pages 227–236, 1993.

[157] H. Zhang and D. Ferrari. Rate-controlled service disciplines. *Journal of High-Speed Networks*, 3(4):389–412, 1994.

Curriculum Vitae

Samarjit Chakraborty

Date of Birth:

December 25, 1972

Place of Birth:

Calcutta, India

Education:

since 11/98
Research assistant and Ph.D. student at the Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Switzerland (Advisor: Prof. Lothar Thiele)

7/96 – 7/98
Graduate studies towards a Master's degree at the Computer Science and Engineering Department of the Indian Institute of Technology Kanpur, India, and also worked as a Teaching Assistant. Graduated with M.Tech. (Master of Technology) in Computer Science and Engineering;
Thesis: *Approximation Algorithms for 3-D Common Substructure Identification in Drug and Protein Molecules*
(Advisor: Prof. Somenath Biswas)

8/92 – 6/96
Undergraduate studies at Jadavpur University, Calcutta, India. Graduated with B.E. (Bachelor of Engineering) in Computer Science and Engineering, and obtained a First Class with Honours.

Practical Experience:

Worked as a Summer Intern at the Indian Statistical Institute Calcutta, India, during the Summers of 1994 and 1995.