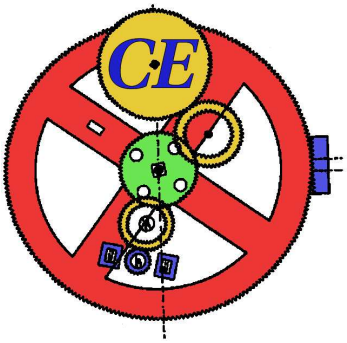


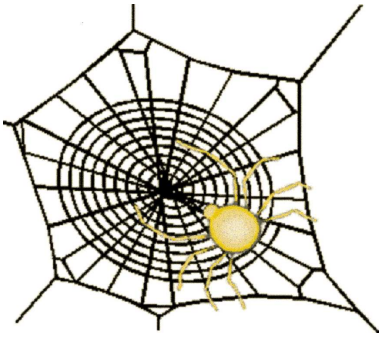
Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl

2003

MSc THESIS

Benchmarking Real-Time Network Processing

Yunfei Wu



CE-MS-2003-01

Abstract

The latest Internet developments pose two requirements on network devices: performance and flexibility. Such requirements have sparked the emergence of the network processor. The advent of network processors has significantly contributed to the efficient management of bandwidth resources, the resolution of latency-related problems, the processing data at wire-speed and the support of emerging applications over Internet. A key challenge is to find an adequate way to evaluate the performance of the heterogeneous collection of network processors when used in a networking environment, such as web switches, routers, etc. It is important and necessary to create benchmarks to evaluate the performance of network processors with different architectures. Existing network processing benchmarks have not covered the processing on real-time (multimedia) data delivery, which is required by numerous new emerging applications, such as Voice over IP (VoIP). VoIP becomes a promising technology because it is inevitable that the convergence of voice and data will play an important roll in future's network. It is becoming increasingly more important to create benchmarks on real-time network processing. In this thesis, we introduce a benchmark suite that allows the performance of real-time network processing to be evaluated. The benchmark suite consists of three benchmarks: an RTP Sender benchmark, an RTP Receiver benchmark, and an RTCP Processing benchmark. For each benchmark, three aspects are specified: function, measurement and environment. The results on the benchmark suite are presented from performance and architectural characteristics points of view. The performance results highlight how fast, measured in terms of clock cycles, the benchmarks are executed. Subsequently, profiling on the benchmarks is performed to determine time-critical functions. The results on architectural characteristics show that the created benchmarks focusing on real-time network processing are justified when compared with existing benchmarks, such as NetBench and MediaBench. The created benchmark suite in this thesis helps to measure and evaluate the performance of network processors and direct the design of the architectures of future network processors.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Benchmarking Real-Time Network Processing

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Yunfei Wu
born in Suixi, China

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Benchmarking Real-Time Network Processing

by Yunfei Wu

Abstract

The latest Internet developments pose two requirements on network devices: performance and flexibility. Such requirements have sparked the emergence of the network processor. The advent of network processors has significantly contributed to the efficient management of bandwidth resources, the resolution of latency-related problems, the processing data at wire-speed and the support of emerging applications over Internet. A key challenge is to find an adequate way to evaluate the performance of the heterogeneous collection of network processors. Benchmarks are needed to evaluate the performance of network processors with different architectures. Existing network processing benchmarks have not covered the processing on real-time delivery, which is required by numerous new emerging applications, such as Voice over IP (VoIP). It is becoming increasingly more important to create benchmarks on real-time network processing. In this thesis, we introduce a benchmark suite that allows the performance of real-time network processing to be evaluated. The benchmark suite consists of three benchmarks: an RTP Sender benchmark, an RTP Receiver benchmark, and an RTCP Processing benchmark. For each benchmark, three aspects are specified: function, measurement and environment. The results on the benchmark suite are presented from performance and architectural characteristics points of view. The performance results highlight how fast, measured in terms of clock cycles, the benchmarks are executed. Subsequently, profiling on the benchmarks is performed to determine time-critical functions. The results on architectural characteristics show that the created benchmarks focusing on real-time network processing are justified when compared with existing benchmarks, such as NetBench and MediaBench. The created benchmark suite in this thesis helps to measure and evaluate the network processors performance and direct the design of architectures of future network processors.

Laboratory : Computer Engineering
Codenumber : CE-MS-2003-01

Committee Members :

Advisor:	Stephan Wong, CE, TU Delft
Chairman:	Stamatis Vassiliadis, CE, TU Delft
Member:	Piet Van Mieghem, NAS, TU Delft
Member:	Ben Juurlink, CE, TU Delft

To my family for their endless love and support

Contents

List of Figures

List of Tables

Acknowledgements

I feel most fortunate to have had the opportunity to study in the Delft University of Technology (TU Delft) and to do my Master's thesis at the Computer Engineering (CE) Laboratory. I would like to express my sincere gratitude to all who gave me the possibility to complete this thesis. Without them, it would have been much harder to finish this thesis.

First and foremost, I am highly indebted to my advisor Prof. Stephan Wong whose help, stimulating suggestions and encouragement helped me in all the time of this thesis. He guided me not to get lost during the development of this thesis. He provided a motivating and enthusiastic atmosphere during the many discussions we had. He was so patient and devoting in correcting my draft, which took him many weekends and holidays. It was a great pleasure to do this thesis under his supervision.

I am deeply grateful to Prof. Stamatis Vassiliadis, a nice person, an excellent teacher and a well-credited scientist, who gave me the opportunity to work on this thesis in the CE group, encouraged me to keep going with my thesis, and gave me a lot of valuable suggestions for this thesis.

I want to say that I really enjoy the company of wonderful people I have met in the CE group, especially in the Room HB15.090. I want to thank Robbert and David for their help on Linux, latex, and, of course, Dutch study.

I want to thank all my friends, especially my special boys and girls in China. I really enjoy the memory of four-year time with them and of their youth and energetic spirit, which kept me "tian tian xiang shang" during the past two years' study in TU Delft.

Finally, my special thanks go to Zhengfei Guan for his long support. The most valuable thing I have acquired from the two years' study in TU Delft is an increased confidence in myself, and an additional faith in my ability to achieve. Nothing would have ever been achieved without his continuous support and encouragement.

Yunfei Wu
Delft, The Netherlands
July, 2003

1 Introduction

In today's high-speed networked world, bandwidth is quickly becoming a critical resource. Due to an explosive growth of Internet usage, a demand arose for network processing both with low latency and at high throughput in order to eliminate network bottlenecks. The advent of network processors has significantly contributed to the efficient management of bandwidth resources, the resolution of many latency-related problems in a broad range of applications, the processing of data at wire-speed, and the support of emerging applications over the Internet. Since there is no generally accepted common implementation of a network processor, many different network processors with different architectures exist. Additionally, each network processor meets varying requirements posed by the wide field of network processing. Therefore, a key challenge is to find an adequate way to evaluate the performance of the heterogeneous collection of network processors (both current and possibly future ones). Similar to the performance evaluation of general-purpose processors, benchmarking is a viable approach to evaluate network processing performance. This approach is not new and several benchmark suites already exist. However, there is still a need for a new set of benchmarks in order to more accurately reflect the constantly evolving field of network processing. More specifically, we introduce a benchmark suite that covers and is tightly related to real-time network processing, which has not been introduced in the existing benchmarks. Next to the obvious performance evaluation purpose, two secondary goals exist for the newly created benchmarks on the real-time network processing. First, the profiling of the newly created benchmarks will provide valuable information to help direct further investigations into acceleration of operations on specialized hardware to be utilized in future network processors. Second, the architectural characteristics of the benchmarks will provide valuable information to understand the features of the real-time network processing, which can be utilized in the design of future network processor architectures.

This chapter is organized as follows. Section 1.1 presents the current situation of network processing together with the observation that real-time delivery services is required by many emerging applications. Section 1.2 introduces the objectives of this thesis by posing some related questions and the methodology described in this thesis. Section 1.3 defines the framework of this thesis.

1.1 Network Processor Implementations

In recent years, we have been witnessing three phenomena of the Internet: a fast-paced growth of the Internet, an explosive increase in network bandwidths, and a dramatic increase of Internet-based applications. Moreover, the witnessed phenomena do not show signs of decline. First, the fast-paced growth of the Internet is driven by the

general popularity of the Internet, the growing need for remote access to information, and emerging applications. Second, the explosive increase of network bandwidths is driven by technological advances in for example fiber optics. Currently, 10Gbps links are well-established and soon broader bandwidth links of 40Gbps or higher will be available. Third, the dramatic increase of Internet-based applications is driven by people putting increasingly more types of data on the Internet with each type requiring different applications to handle them.

The latest Internet developments give rise to two requirements posed on network devices: performance and flexibility. Performance is needed because network devices are expected to process at wire-speed in order to eliminate network bottlenecks. Flexibility is needed because network devices are expected to be easily adaptable in order to support emerging applications. In designing network devices for flexibility, an obvious choice would be to utilize general-purpose processors (GPPs). The programmable GPPs have the flexibility to be adapted to rapidly changing network protocols. However, they usually lack in performance to handle data at wire-speed. Traditionally, indeed GPPs were used to design network devices. As the performance became increasingly more important, application-specific integrated circuits (ASICs) were introduced. However, ASICs lack the flexibility to be easily updated in order to support new features. Consequently, a new kind of processor is needed to meet the two requirements at the same time. Such a need has sparked the emergence of the network processor (NP).

Generally, a network processor is a programmable device incorporating specialized hardware designed specifically to process data at wire-speed. Network processors can provide speed through architectural improvements, such as parallel distributed processing and pipeline processing designs. The programmability of network processors can enable easier migration to new protocols and technologies. While network processors are designed having both flexibility and performance in mind, there is still a broad spectrum of trade-offs between these two requirements. The advantage is that it allows network processor vendors to distinguish their products from others by targeting slightly different application areas. The main disadvantage is that the existence of many architectures and implementations of network processors makes it difficult to compare them in terms of performance. Consequently, benchmarks are needed to measure the performance of network processors. Simply put, a benchmark is a standard for judging system performance in a target application. Currently, several network processing benchmarks have been introduced. Although some network processing benchmarks exist, there is still headroom for benchmarking network processing due to the two facts. First, existing benchmarks do not cover all applications in the networking domain. It is necessary for benchmarks to cover some of the untargeted applications. Moreover, with new applications emerging , no benchmark exists to evaluate the performance of network processors on these new applications. Second, existing benchmarks mainly cover lower layer[1] functionalities while there is a certain trend that higher layer functionalities must also be supported, for example, real-time delivery of multimedia data.

[1]Communication models (e.g., the TCP/IP model or the OSI model) are usually layered in order to describe and implement a complex task that is communication between computers.

In this thesis, we describe the implementation of a new benchmark suite that specifically incorporates higher layer functionalities. Two additional benefits arise from the created benchmarks. The first benefit is to perform profiling on these benchmarks in order to gain more insight in such higher layer functionalities. Such insight can be utilized to implement specialized hardware in the design of future network processors. The second benefit is to investigate architectural characteristics on such higher layer functionalities in order to understand the features of these functionalities and their performance. These characteristics can be taken into consideration in designing the architectures of future network processors. The next section discusses in detail the questions investigated in this thesis and the methodology described in this thesis.

1.2 Open Questions and Methodology of Thesis

This study is driven by two trends that have been identified in network processing. The first trend is that the shift in supporting NP functionalities is from lower layers to higher layers of the TCP/IP model. It is becoming increasingly more important to create benchmarks that allow the investigation of higher layer functionalities. The second trend is in the emergence of applications that require real-time delivery. One of such applications is Voice over IP (VoIP). As the need of multimedia services grows, it is inevitable that the convergence of voice and data will play an important roll in future's network. Therefore, VoIP is becoming a promising technology. However, there are some challenges in utilizing the IP network to transfer voice data with real-time characteristics. In order to ensure real-time delivery, specific processing is needed.

Considering the two trends together with the fact that previously introduced network processing benchmark suites mainly focus on the lower layer functionalities, we believe there is a need for a new set of benchmarks that incorporate higher layer functionalities. The first question investigated in this thesis is that:

1. *How can we create benchmarks that allow the investigation of real-time network processing?*

As mentioned, the newly created benchmarks incorporates many functionalities related to real-time delivery. This opens up the opportunity to perform profiling on the newly created benchmarks in order to find the time-critical functions. By implementing these time-critical functions on specialized hardware, performance gains can be achieved. The second question investigated in this thesis is that:

2. *Which functions in the newly created benchmarks on real-time network processing are time-critical?*

The created benchmarks also allow us to investigate architectural characteristics in the real-time delivery processing that may be utilized in the design of future network processor architectures. The investigation entails the following four characteristics: instruction level parallelism, branch prediction accuracy, instruction distribution, and cache behavior. First, a high instruction level parallelism means that it can be and should be

exploited in the processor design. Second, a high branch prediction accuracy means that less exotic complex branch predictors can be utilized, e.g., to keep the processor pipeline filled with useful instructions. Third, depending on the instruction distributions, design decisions can be made whether to include certain functional units or not. Fourth, since both the instruction cache and the data cache can have a profound effect on performance, it is important to understand their behavior. The third question investigated in this thesis is that:

3. What are the main architectural characteristics of real-time network processing?

The answer to the third question can also highlight the main differences between the characteristics of real-time network processing with other processing or other network processing. This thesis will mainly focus on the investigation on these three questions posed above and achieve the final answers to these three questions.

The methodology that has been used in order to achieve the answers to the three mentioned questions is briefly described as follows:

- **The benchmarks** are implemented by taking the existing C code and modifying it. It is not the intention of the MSc project to develop new applications or application codes in that matter. Since we focus on real-time network processing, we have chosen to focus on both the Real-Time Transport Protocol (RTP) and the Real-Time Transport Control Protocol (RTCP). In implementing the benchmarks, we specify three aspects: function, measurement and environment (interfaces and simulation).
- **The time-critical functions** are determined by the profiling information of the benchmarks. The profiling is performed by running the benchmarks in a simulation environment that allows performance metrics (e.g., clock cycles) to be measured.
- **The architectural characteristics** are performed by again running the benchmarks in a simulation environment. Furthermore, the determined characteristics are compared to results from other benchmarks, such as NetBench and MediaBench.

It must be noted that the work described in this thesis does not entail any actual hardware design. The work solely focuses on obtaining software benchmarks, profiling information, and architectural characteristics by utilizing an appropriate simulation environment.

1.3 Thesis Framework

This section discusses the framework of this thesis that consists of the following chapters:

Chapter 2 reviews the background of this thesis from three aspects. First, it discusses the 4-layer TCP/IP model, presents the different responsibilities and some key characteristics of each layer, highlights the underlying functions in network processing, and

proposes a number of solutions to design embedded processors in networks. Second, it discusses a benchmark methodology for network processing and highlights some existing network processing benchmarks. Third, it presents a general overview of the SimpleScalar tool set used in order to obtain simulation results.

Chapter 3 analyzes the differences between real-time delivery and non-real-time delivery, describes RTP/RTCP, i.e, the protocols used for providing real-time delivery, and introduces the implementation on the benchmarks for RTP/RTCP processing. The created benchmarks are described from three aspects: function, measurement and environment.

Chapter 4 discusses the simulation results on the benchmarks for RTP/RTCP processing. The benchmarks are performed on the *sim-outorder* simulator from the SimpleScalar tool set (version 3.0). The results are analyzed from two aspects. First, profiling results based on the clock cycles are performed in order to find the time-critical functions in the benchmarks. Second, the architectural characteristics results are presented and compared with NetBench and MediaBench.

Chapter 5 presents the conclusions of this thesis, describes the main contributions of this thesis and highlights several future research directions.

2 Background

The advent of network processors (NPs) made a significant contribution to maximizing the utilization of bandwidth and support of emerging internet-based applications. NPs can provide intelligent network processing with low latency and high throughput at wire-speed. One cannot appreciate the technical details of underlying functions in network processing without understanding the TCP/IP model[1]*. It is the fundamental model on which the current Internet is based.*

This chapter is organized as follows. Section 2.1 briefly reviews the TCP/IP model, explains major protocols used in this model, and highlights characteristics of the Internet. Section 2.2 presents several underlying functions in network processing and possible architectures for network processing. Section 2.3 poses the question why benchmarking is needed, analyzes the characteristics of a benchmark, presents a benchmarking methodology for network processing, and lists some existing network processing benchmarks. Section 2.4 briefly reviews the SimpleScalar tool set, which includes several simulators for evaluating system performance. Section 2.5 presents the conclusions of this chapter.

2.1 The TCP/IP model

A protocol is a set of rules that governs data communication between devices. It defines what is communicated, how it is communicated and when it is communicated. Due to the complexity of communication between two different devices, network protocols are normally developed in *layers*, with each layer responsible for a different facet of the communication. Two important benefits result from decomposition into layers. First, the different layers can be designed more or less independently, and therefore greatly simplifying network design. Second, compatibility is derived from the independent inner-working of each layer in the sense that each layer can be regarded as a black box with its own defined inputs and outputs. TCP/IP is normally considered to be a *4*-layer system, as depicted in Figure 2.1 [23]. Figure 2.2 depicts the layered principle in the TCP/IP model which entails that layer n at the destination receives the same object sent by layer n at the source. Figure 2.2 also illustrates how data are sent from computer A to computer B. As the data travel from computer A to computer B, it may pass through many intermediate nodes, such as routers and switches. These nodes usually involve only the first two lower layers of the TCP/IP model. The transmission first starts at the application layer in computer A and then moves down through the layers of computer A. At each layer n, a header Hn is added to the data unit received from the next higher layer, e.g., $H4$ is the header added at the layer *4*. At the layer *1*,

[1]The TCP/IP model is a protocol suite which combine different protocols in various layers.

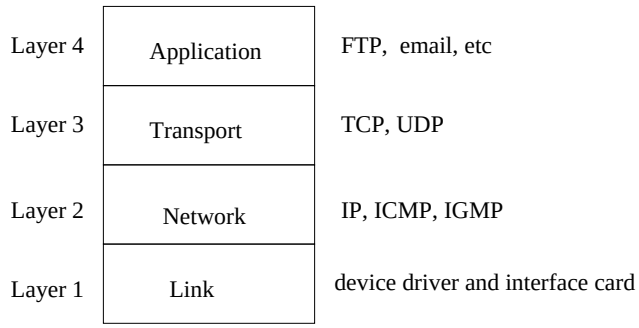


Figure 2.1: The four layers of the TCP/IP protocol suite.

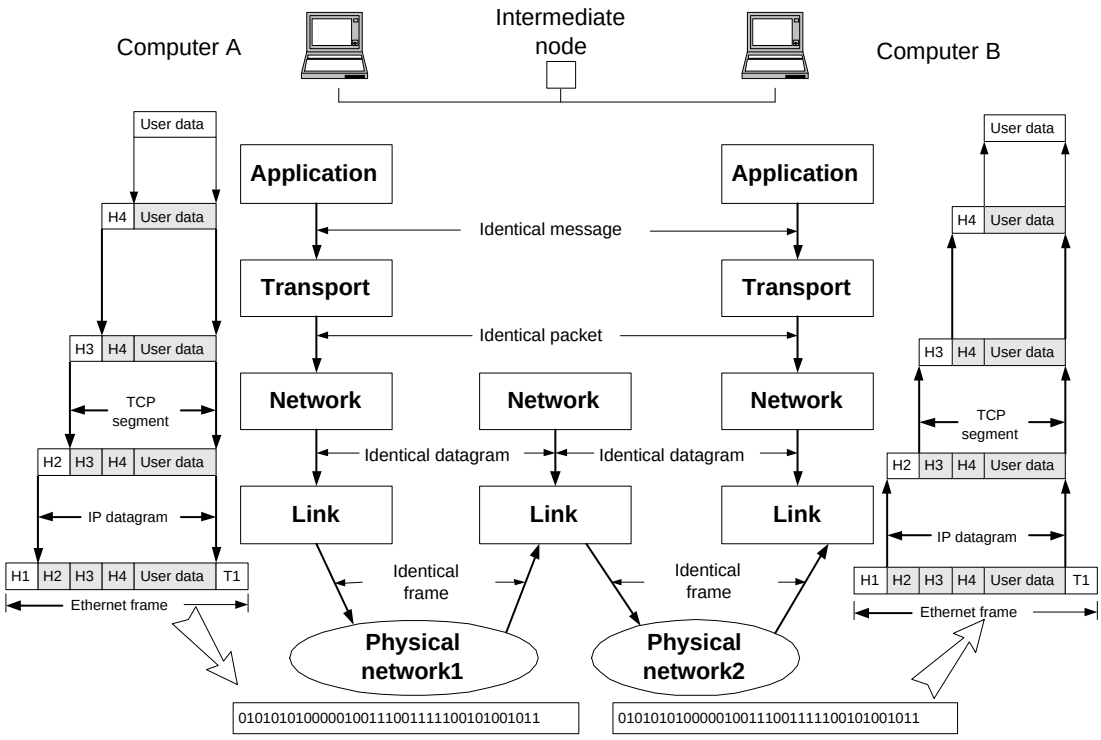


Figure 2.2: Illustration of data transmission from computer A to computer B.

a trailer $T1$ is added as well. Finally, when the formatted data unit passes through the physical network, a stream of bits are sent out. Upon reaching the destination, the data then move upwards through the layers of computer B. At each layer, the previously attached headers and trailers are removed.

The different responsibilities and some key characteristics of each layer are discussed as follows:

1. The *link layer*, also called the *data-link layer* or the *network interface layer*, normally includes the device driver in the operating system and the corresponding

network interface card in the computer. The network interface is assigned a unique physical address, e.g., *48*-bit physical address for an Ethernet card, by which the computer is recognized. This layer is responsible for sending and receiving data from the network layer, communicating with the network interface in transferring data, and handling all the hardware details of physical interface. TCP/IP supports many different link layers, depending on the type of networking hardware being used: Ethernet, Token ring, Fiber Distributed Data Interface (FDDI), and the like. The data unit generated at the link layer is called a frame by adding a header and a trailer information - Cyclic Redundancy Check (CRC) - to the data received from the network layer. The maximum length of the frame is Maximum Transmission Unit (MTU) which limits the number of bytes of data that can be transferred. For Ethernet, its MTU is *1500* bytes. In this layer, two main protocols are defined: the Address Resolution Protocol (ARP) and the Reverse Address Resolution Protocol (RARP). ARP maps an IP address to a physical address while RARP performs the reverse operation.

2. The *network layer*, also called the *internet layer*, is responsible for packaging, addressing, and routing functions. The core protocol of this layer is the Internet Protocol (IP), which is one of the major protocols of the TCP/IP protocol suite (TCP being the other). The basic data object in this layer is the IP datagram, which is a variable-length packet consisting of two parts: a header and a data unit. The header can be from *20* to *60* bytes and contains information, e.g., IP address that is essential to routing and delivery. The most fundamental service provided by IP is a packet delivery. However, this service is unreliable, best-effort, connectionless. The term *unreliable* means that packet may be delayed or duplicated at the destination. Or, the packet may be lost and cannot reach the destination. *Best-effort* means that IP delivers the packets without detecting problems or informing the sender or receiver about the problems. *Connectionless* means that IP treats each datagram as an independent entity unrelated to any other datagrams. In other words, IP datagrams can go through different paths to the destination. Therefore, out of order delivery is possible.

   The network layer also contains two other protocols, Internet Control Message Protocol (ICMP) and Internet Group Management Protocol (IGMP). ICMP is responsible for exchanging error messages and other vital information with the network layer in another host or route. IGMP is responsible for multicasting: sending a UDP datagram to multiple hosts.

3. The *transport layer* provides communication from one application on computer A to another on computer B, which is called end-to-end communication. This layer contains two main protocols: Transport Control Protocol (TCP) [13] and User Datagram Protocol (UDP) [14]. Both TCP and UDP provides a port number to higher layer for applications to identify the endpoints of the communication. The data unit that TCP sends to IP is called a TCP segment. The one that UDP sends to IP is called a UDP datagram.

   TCP provides a one-to-one, connection-oriented, reliable communication service.

The term *connection-oriented* means the application using TCP must establish a TCP connection between the two end systems before they can exchange data. TCP uses *positive acknowledgment* (ACK) with *retransmission* in order to provide a reliable service. Positive ACK means that the receiver will send back an ACK message to the sender when it receives the data. If the ACK is not received by the sender within a timeout interval, the data is retransmitted. At the receiver, the *sequence number* in the TCP header is used to correctly order the segments that may have been received out of order and to eliminate duplication. Packet damage is handled by adding a *checksum* to each transmitted TCP segment, checking it at the receiver and discarding it if it is damaged.

UDP provides a one-to-one or one-to-many, connectionless, unreliable communication service. UDP is much simpler than TCP for three reasons. First, UDP does not use acknowledgements to ensure packet delivery. Secondly, it does not provide feedback to control the rate at which information flows between the two end systems. Thirdly, it does not order incoming data. Therefore, UDP datagrams can be lost, duplicated or arrive out of order. Furthermore, packets can arrive faster than the receiver can process them. Normally, an application program that utilizes UDP takes full responsibility for reliable packet delivery, which includes message loss, duplication, delay and out-of-order delivery. The header length of UDP is also much smaller than TCP header length. Therefore, UDP is often utilized when the amount of data to be transferred is small (such as the data that would fit into a single packet), when the overhead of establishing a TCP connection is not desired or when the applications or upper layer protocols provide reliable delivery.

4. The *application layer*, the highest layer, contains application programs that provide services across a TCP/IP network. The applications includes File Transfer Protocol (FTP), email, Voice over IP (VoIP), video conferencing, etc. An application interacts with one of the transport layer protocols to send and/or receive data.

In conclusion, the TCP/IP model is used and tested extensively in the Internet. it contains many protocols, each of which provides a specific functionality. Moreover, newer protocols are emerging in this model in order to meet the need of the potential applications.

2.2 Network Processing

Generally, one cannot design a system without understanding the requirements of the application that the system targets. Therefore, it is not possible to design the network processing architectures without understanding its underlying functions. This section discusses the most important underlying functions in network processing first, and subsequently describes possible architectures for network processing.

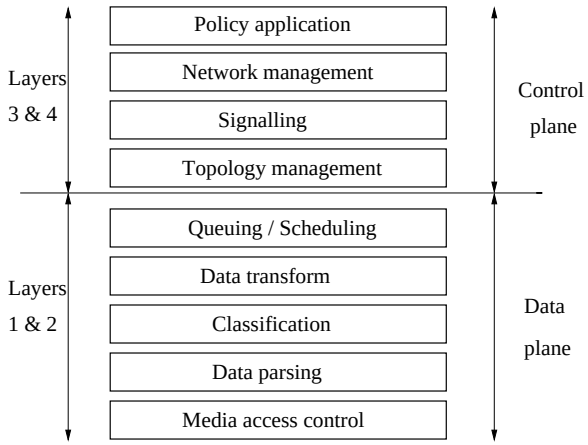


Figure 2.3: Processing tasks in data plane and control plane.

2.2.1 Underlying Functions

The main underlying functions in network processing focus on two planes: the data plane and the control plane. Both planes are depicted in Figure 2.3 [11]. In relation to the TCP/IP model, processing in the data plane focuses on layers *1* and *2*, while processing in the control plane focuses on layers *3* and *4*.

Network processing in the data plane concerns analyzing and modifying each incoming packet, managing an input/output queue, scheduling packets in the queue, and forwarding the packets towards their destinations. Packets must be processed at wire-speed. The underlying functions in the data plane (Figure 2.3) are discussed as follows:

- *Media Access Control* (MAC) [4] is a general reference to the low-level hardware protocols used to access a particular network . The term *MAC address* is often used as a synonym for *physical address*. Normally, the hosts and routers are recognized by their own physical addresses. However, physical addresses are not adequate in an inter-networking environment where different networks can have different address formats. A universal addressing system in which each host can be identified uniquely, regardless of the underlying physical network, is needed. The IP address is designed for this purpose. This means that delivery of a packet to a host requires both two types of addressing: a physical address and an IP address. Therefore, the mapping between an IP address and a physical address and vice versa are needed, which is done by ARP and RARP, respectively.

- *Data parsing* and *classification* look at a packet header and classify the packet based on a set of rules. Both functions can be found in all layers. For example, in the web server application, a packet to this server is classified by two rules: the destination port in the TCP header must be *80* and the protocol in the IP header must be *6*. All packets that do not match the two rules cannot reach the web sever.

- *Data transformation* entails many different operations: changing the content of a

packet by adding additional information to the packet, modifying the packet when passing through different networks and segmenting, fragmenting and reassembling the packet. These operations differ significantly in complexity. For example, a packet may be encrypted by using some encryption algorithms for security purposes. Or, a packet may be encapsulated as a payload in a new packet with a new header when the packet passes from an IP network to an ATM network.

- *Queuing and scheduling* manage an input buffer and an output buffer, make the decision on how to insert packets to queues and dequeue packets, and schedule packets for different applications. The packets are scheduled by many different policies and their priorities.

Network processing in the control plane controls and manages device operations, updates routing table, executes routing algorithms, and processes some exception packets passed from the data plane. Unlike processing in the data plane, processing in the control plane will not process all packets. Therefore, it is less time-critical. The underlying functions in control plane are discussed as follows:

- *Topology management* and *network management* consist of the following operations: monitoring network activity, distributing a database, generating real-time graphical views of the network topology, and updating a dynamic routing table. A routing table is used to forward a packet to the destination or to the next hop. It has to be updated as soon as there is a change in the Internet. For example, when a router is connected to the Internet for the first time, the routing tables of all routers on the Internet need to be updated by using routing protocols, such as the Routing Information Protocol (RIP) or the Open Shortest Path First (OSPF) protocol.

- *Signalling* ensures quality of services since the IP-based network only provides "best-effort" services. Two groups have proposed signalling standards for IP telephony. The International Telecommunication Union (ITU) has defined a suite of protocols known as *H.323*, and the Internet Engineering Task Force (IETF) has proposed a signalling protocol known as the *Session Initiation Protocol* (SIP). Some other signalling protocols, such as *Resource ReserVation Protocol* (RSVP) and *Multi-Protocol Label Switching* (MPLS), are also utilized in IP networks.

- *Policy application* is a set of rules which are used to execute routing algorithms and to update routing table. Considering a packet forwarding example, IP will perform a table lookup in order to determine the next hop where the packet should be forward.

A comparison between data plane processing and control plane processing is presented in Table 2.1 [11] [6]. There are five main differences between data plane processing and control plane processing. Because three differences out of five (in the first three rows of Table 2.1) are already discussed above, only the remaining two differences are discussed here. First, the difference in the software is that normally data plane processing is implemented in specific hardware written in VHDL language while control plane processing is implemented in software written in C or C++. Second, the difference in execution is

	Data plane	Control plane
Main functions	Packet forwarding	Routing and Signalling
Layer	In layer 1-2	In layer 3-4
Performance	Real-time processing	Less time-critical
Software	Fewer than 1500 lines of code and typically written in VHDL or low-level microcode	Millions of lines of code and written in C/C++
Execution	On a special processor accelerated for maximum performance	On a general-purpose CPU

Table 2.1: Comparing data plane processing with control plane processing.

that the data plane processing is launched on a special processor in order to achieve high performance while the control plane processing is launched on a general-purpose processor in order to achieve flexibility. With the increased demand for complex processing, the boundaries between the data plane and the control plane processing have become blurred.

2.2.2 Network Processing Design Alternatives

A complete design methodology [24] - a design process - for embedded computing systems tells us that we need to consider the major goals of the design when creating a product, which may include functionality, performance, power consumption and manufacturing cost. Functionality provides a more detailed description of what the product does; Performance means the processing speed of the product, which may be a combination of soft deadlines such as approximate time to perform a user-level function and hard deadlines by which a particular operation must be completed. Power consumption gives a rough idea of how much power the product can consume. And manufacturing cost primarily defines the cost of the hardware components. Beyond this, we must also consider some other important requirements in system design: time-to-market, design cost and quality.

In designing network devices, the latest developments in the Internet give rise two requirements: performance and flexibility. Traditional network processing mainly focuses on forwarding packet at high speed in order to eliminate the network bottlenecks. Network devices are expected to perform at high speed with low latency. However, as the Internet Protocol keeps maturing, newer protocols have been emerging and will emerge in the future. Such newer protocols include security, signalling, and various network managements, etc.. Network devices are also expected to flexibly support these newer protocols and applications with high performance. In general, there are a number of techniques for designing network processors:

- *General-purpose processor (GPP)* - A programmable processor for general-purpose

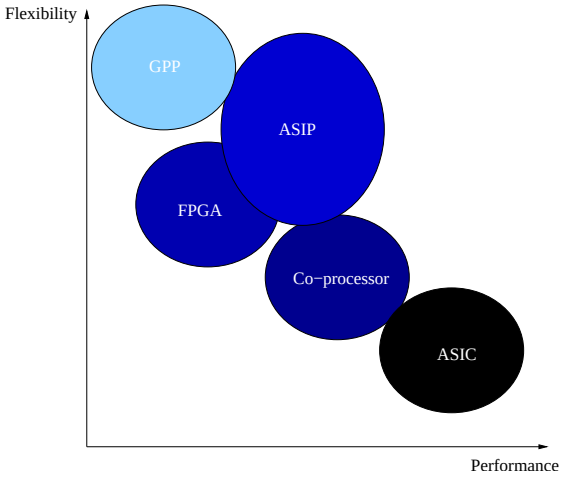


Figure 2.4: The space of system implementations [21].

computing.

- *Application-specific integrated circuit (ASIC)* - An accelerator with specific function units. It interacts with the CPU through the programming model interface. It does not execute instructions.
- *Co-processor* - A separate chip connected to the internals of the CPU or the same chip nowadays. It implements additional instructions, which are not supported by the main processor to which it is attached to.
- *Application-specific instruction processor (ASIP)* - A CPU whose instruction set has been optimized for a particular set of applications.
- *Field-programmable gate array (FPGA)* - A programmable logic chip that can be quickly customized to a particular function. FPGAs come in two basic varieties: one-time-programmable devices which can be programmed once and reprogrammable devices which can be programmed many times with new logic designs.

In order to examine which technique can meet the two requirements, Figure 2.4 maps these categories on two axes: flexibility and performance. In Figure 2.4, We can see that GPP is the most flexible at the cost of the performance and ASIC provides the best performance with the worst flexibility.

Earlier network devices, such as routers and switches, employ GPPs for network processing. The best advantage of this approach is that it enables flexible support new features required by the new applications since GPP is a programmable processor for general purpose computing. However, the drawback of this approach is its limited ability to scale the performance in order to support the demands for higher bandwidths. With bandwidth increasing from OC-3 (*155Mbps*) to OC-192 (*10Gbps*) and up to OC-768 (*40Gbps*) or higher in the future, design priority shifts from flexibility to speed. GPPs can not have enough performance to process data at wire-speed. Therefore, the market

turned to high-performance ASICs for packet processing. An ASIC does accommodate the increasing demand for speed when performing the required packet processing. However, it is restricted by its inherent inflexibility. Nowadays, as mentioned before, since security, QoS and multimedia applications become the new networking concerns, ASICs are limited to these functionalities and can not support new features.

In conclusion, the flexible combinations of the data plane and the control plane processing with high performance are not possible with either ASIC or GPP solutions alone. Today, the market is turning to a new solution, network processors (NPs): high-performance, programmable devices designed to efficiently execute communications workloads [6]. A network processor is an ASIP for the networking application domain. Figure 2.4 also depicts that ASIP is the best flexibility-versus-performance trade-off for networking system implementations. Currently, there are a number of network processors available in the market, such as PayloadPlus network processor from Agere, Toaster2 from Cisco, PowerNP network processor from IBM, and IXP2400 network processor from Intel and so on. More details about the architecture of each network processor can be found in [21] and [6].

2.3 Benchmarking

Before we discuss benchmarking in relation to network processing, we present its general definition. A benchmark[2] is:

1. something that serves as a standard by which others may be measured or judged.

2. a standardized problem or test that serves as a basis for evaluation or comparison (as of computer system performance).

Generally, a good benchmark helps to measure system performance in a deterministic and reproducible manner. From a marketing point of view, we know that reliable performance equates to excellent customer services, while excellent customer services equate to maximized profits. Therefore, it is important for benchmarks to evaluate the performance of network processors. When referring to more general-purpose processors, an immediate term comes into mind, namely SPEC that stands for Standard Performance Evaluation Corporation [22]. Its mission is to establish, maintain, and endorse a standardized set of relevant benchmarks and metrics for performance evaluation of modern computer systems. However, The benchmarks defined in SPEC are not suitable for benchmarking network processors since network processors target to specific applications related to networking. This section presents a literature study in benchmarking network processors: a benchmarking methodology for network processor, a framework for benchmarking network processor and an overview of currently existing several benchmarks for network processors.

[2]The definition is taken from www.m-w.com.

2.3.1 A Benchmarking Methodology

According to [6], a good benchmark must possess three characteristics: accuracy, scalability, and representativeness. First, benchmarks must accurately reflect real-world applications. Additionally, they can accurately measures system performance in a target application. Second, benchmarks must be able to test different data sets without requiring major changes. In addition. they must be comparable across system implementations. Third, benchmarks must accurately represent a system-under-test, cover most functionalities of one domain, and provide results that correlate to real-world performance.

In [17], [12], [18] and [7], several benchmarking methodologies in the networking domain are discussed in detail. Due to the heterogeneity of network processor architectures, it is necessary to separate benchmarking concerns to ensure comparability of results. [6] defines the benchmarking methodology for network processors from three separate specifications:

- **Functional specification** defines the requirements and constrains of a benchmark , such as the minimum allowable routing table size supported by the implementation and so on. It describes the core algorithmic behaviors of the benchmark.
- **Measurement specification** defines the performance metrics for evaluating the benchmark performance, such as latency, throughput, and packet loss ratio. It also specifies the format in which to present the results and the procedures used to measure the metrics.
- **Environment specification** specifies the network interface and the control interface. It defines the test configuration used to obtain the metrics, and input/output behaviors being benchmarked.

These three specifications present the precise specifications of a benchmark and allow to produce the benchmark results with accuracy, scalability and representativeness. This methodology allows to measure the performance of network processors with different architectures.

2.3.2 Framework for Benchmarking Network Processors

As mentioned before, network processing performs a variety of tasks, some in the data plane and some in the control plane. Until now there are a number of network processors available from different vendors with different architectures. This makes benchmarking network processors more complex. In order to cover different network processor architectures, to measure the complete network processing space, and to meet the needs of different users, [6] and [15] categorize benchmarking into 4 levels:

- **System level** Benchmarks at this level are used to measure the performance of complete systems, such as firewalls, multi-service switches, etc. Both the data plane processing and the control plane processing are evaluated. A set of system-level

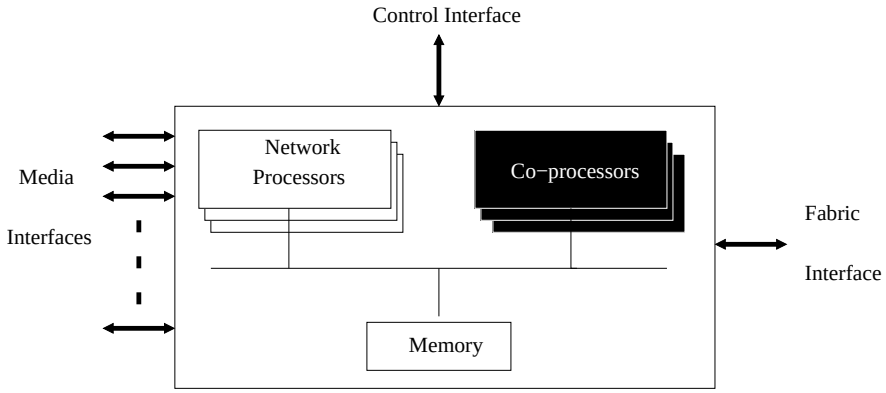


Figure 2.5: A reference platform for application-level benchmarks [15].

benchmarks has already been defined by the IFEFT and Internet Service Provider (ISP).

- **Function level** Benchmarks at this level are used to measure the performance of individual network processing functionalities only in data plane, such as IP forwarding [18], L2 switching [12] and Network Address Translation (NAT). The Network Processing Forum (NPF) has defined a typical reference model depicted in Figure 2.5 for test configuration the benchmarks in this level. This reference model includes one or more network processors, and external data and control interfaces. One or more media interfaces are used through which network data is injected. A control interface is used for control plane operations. A fabric interface is used to forward network data out.

- **Micro level** Benchmarks at this level are intended to measure elementary, stand-alone functions which is not easily decomposed into other functions, such as checksum calculation for error detecting, table lookups used for routing and encryption for security.

- **Hardware level** Benchmarks at this level are focused on the measurement of latencies and throughputs for accessing different hardware resources within network processor, such as memory, input/output interfaces, and other computation elements. Hardware-level benchmarks are architecture specific, that is to say, they can only be defined in terms of a given architecture.

Benchmark suites developed at these four levels have a hierarchical relationship with each other. Typically, a system-level benchmark may include one or more function-level benchmarks; A function-level benchmark may include one or more micro-level benchmarks. For example, a firewall includes IP forwarding, NAT application and some other applications. While IP forwarding includes table lookups, checksum calculation and some other functions.

2.3.3 Existing Network Processing Benchmarks

Recently some benchmarks for network processors [9], [1], [8], [15] and [16] have been discussed.

- NetBench [9] presents a set of nine representative applications in network processing domain. It categorizes these applications into three levels: micro level, IP level and application level. Netbench applications take an IP header trace as the input, simulate the processing on *10000* IP packets by the SimpleScalar simulator, and compare the results with MediaBench, a benchmark for evaluating and synthesizing multimedia and communications systems [3]. NetBench experimental results include instruction level parallelism (ILP), branch prediction accuracy (BPA), instruction distribution (ID) and cache behavior (CB). It also performs these simulations using Inter IPX simulator and compares the performance of IXP2000 with a general-purpose processor.

- CommBench [1] focuses on operations that are performed at the network layer, and includes eight applications which are divided into two groups: header processing applications (HPA) and payload processing applications (PPA). Finally, it compares the results with those of SPECs on code and kernel sizes, computational complexity, instruction frequency and cache performance.

- Embedded Microprocessor Benchmark Consortium (EEMBC) [8] defines three application benchmarks in the network processing domain: routing protocol, Open Shortest Path First (OSPF) based on the Dijkstra algorithm, route lookup and packet management.

- Network Processing Forum (NPF) [15] defines a framework for benchmarking network processors. It categorizes applications in the network processing domain into three levels, which are same with Section 2.3.2 without the hardware level, and mainly focuses applications on the function level.

- Intel Corporation [16] focuses on benchmarking the Intel IXP1200 network processor. The methods for performance measurement and the consideration of interface issue are specific to the IXP1200. They provide some results for IPv4 forwarding on the IXP1200.

These benchmarks mentioned above mainly focus on some specific network processing. They do not cover all applications in the network processing domain. with new applications emerging, there are still no benchmarks to evaluate the performance of the new applications.

2.4 The SimpleScalar Tool Set

This section briefly reviews the SimpleScalar tool set [2] which consists of a compiler, an assembler, a linker, a simulator, and visualization tools for the SimpleScalar architecture. Subsequently, a general overview of the SimpleScalar architecture is presented. Finally,

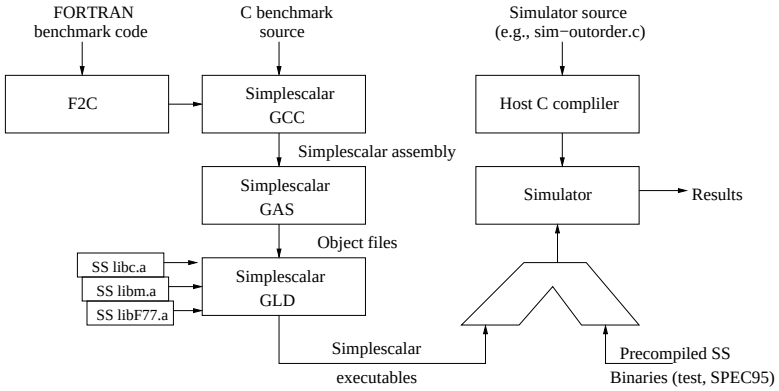


Figure 2.6: The SimpleScalar tool set overview.

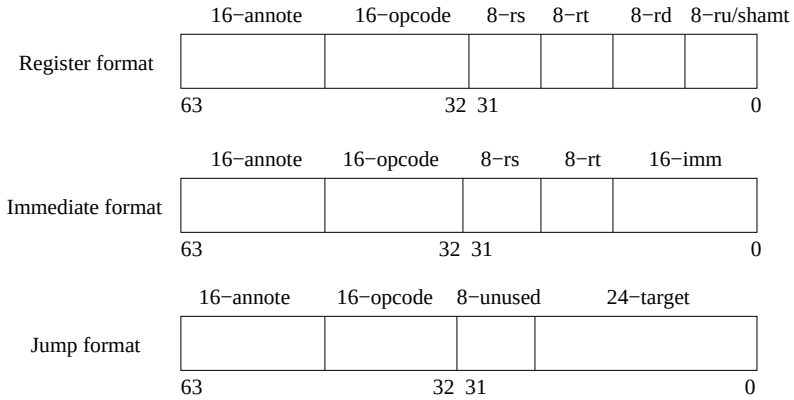


Figure 2.7: The SimpleScalar architecture instruction formats.

some characteristics of the *sim-outorder* simulator are discussed, which is utilized for simulating the benchmarks defined in the next chapter.

Figure 2.6 depicts a graphical overview of the SimpleScalar tool set. A C benchmark before being simulated must be compiled using the SimpleScalar GCC (current version *2.7.2.3*), which generates the SimpleScalar assembly. By using the SimpleScalar assembler, linker and related libraries included in the tool-set, a SimpleScalar executable is generated and can be fed directly into one of the provided simulator, for example, *sim-outorder*.

The SimpleScalar architecture is derived from the MIPS_IV architecture. All Instructions in the SimpleScalar are extended to *64* bits and three instruction encodings are used (depicted in Figure 2.7). The register format is used to computational instructions. The immediate format supports the inclusion of a *16*-bit constant. The jump format supports specification of *24*-bit jump targets. The *16*-bit annotate is used to synthesize new instructions and annotations without changing and recompiling the assembler.

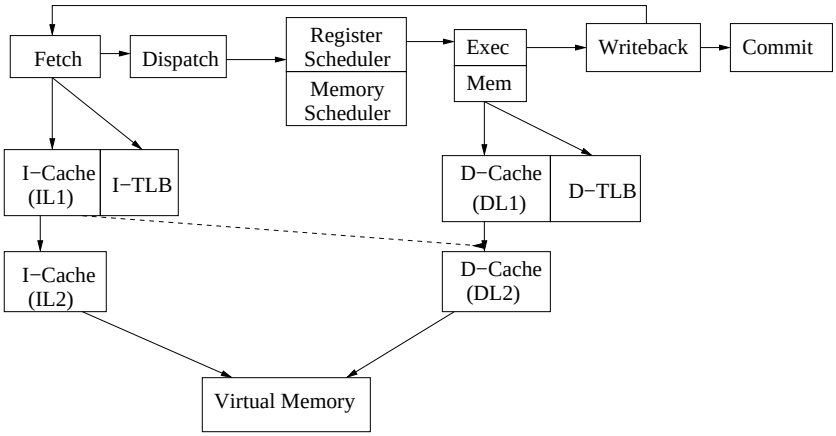


Figure 2.8: Pipeline for the *sim-outorder* simulator.

The most complicated and detailed simulator in the SimpleScalar package is the *sim-outorder* simulator. It keeps track of the timings of events, simulates everything that happens in a superscalar processor pipeline, including out-of-order instruction issue, the latency of the different execution units, the effects of using a branch predictor, etc. Figure 2.8 depicts the *6*-stage pipeline of *sim-outorder*. First, the fetch stage takes the program counter, the predictor state and mis-prediction detection from the branch execution units as inputs. It fetches instructions and sends them to the instruction fetch queue. Second, the dispatch stage takes as many instructions as possible from the fetch queue and sends them to the scheduler queue. It updates the register update unit (RUU) and load/store queue (LSQ), and renames registers. Third, the scheduler stage locates instructions with all register inputs ready, and locates loads with all memory inputs ready. Instructions will be issued to the functional units and its state is updated. Forth, the execute stage takes instructions ready to execute, functional unit state and D-cache state as inputs. It schedules writeback events, and updates functional unit and D-cache state. Subsequently, the Writeback stage gets finished instructions specified by the event queue, and walks the dependence chain of instruction outputs to mark instructions that are dependent on the completed instruction. It also detects branch mis-predictions to the fetch stage. Finally, the commit stage models in-order retirement of instructions, stores commits to the D-cache, and handles D-TLB miss.

2.5 Conclusions

This chapter provided four backgrounds on four topics of this thesis. First, the background on the TCP/IP model discussed the model and main protocols in this model, and presented characteristics of the Internet. Second, the background on the network processing categorized the network processing into two planes, the data plane and the control plane, and described the different processing in these two planes. With the increase of network bandwidth and of network applications, network processing with high performance and flexibility is becoming an important concern. The background on the

network processing presented possible architectures for network processing. Third, the background on benchmarking discussed what, why and how aspects are related to the benchmarking, and listed some existing network processing benchmarks. Finally, the background on the SimpleScalar tool set provided an overview of the tool set, discussed its architecture, and presented the *sim-outorder* simulator.

3 Benchmarking

Previous *chapters reviewed the TCP/IP model and the main underlying functions in network processing. It was argued in the previous chapters that it is important for benchmarks to evaluate the performance of network processors. Furthermore, there is a certain trend that functionalities supported by network processors are shifting from lower layers to higher layers. Therefore, it is more important to benchmarking network processing in higher layers. Currently, Voice over IP (VoIP) has been viewed as an attractive and effective technology. VoIP entails transferring voice data over an IP network. Its popularity stems from two facts. First, as the need for multimedia services grow, the need for Internet to transfer voice together with data becomes inevitable. Second, the IP protocol and associated protocols discussed in Chapter 2 have been universally existed in user and network equipment. The universal presence of IP makes it a very convenient platform from which to launch multimedia data. However, IP only provides "best-effort" services causing the variable delays, packet duplicates, and packet losses. This is usually not a big problem for data applications. While voice packet delivery is a kind of real-time service, which means voice packets have to be delivered timely to eliminate packet delays and provide QoS.*

This chapter focuses on the delivery of real-time data over an IP network. This chapter is organized as follows. Section 3.1 discusses the protocols used to deliver real-time data. Section 3.2 examines the detailed implementation on benchmarks for the protocols. Section 3.3 presents the conclusions of this chapter.

3.1 Real-Time Delivery

As mentioned in Chapter 2, IP was developed to forward non-real-time data. However, VoIP is classified as a real-time service, because it requires timely delivery of voice data. There are two differences between real-time delivery and non-real-time delivery.

- **Tolerance for errors** Real-time delivery exhibits a high tolerance for errors while non-real-time delivery has a low tolerance for errors. For example, if some data packets constituting an email are lost, the email might lose important information rendering its uselessness. However, if some packets constituting a voice conversation are distorted, the fidelity of the voice reproduction is not severely affected.

- **Tolerance for delays** Non-real-time delivery exhibits a high tolerance for delay while real-time delivery has a low tolerance for delay. If some data packets constituting the email are delayed, they are still useful as long as they are delivered to the destination. For emails, the difference between *10* seconds and *1* minute

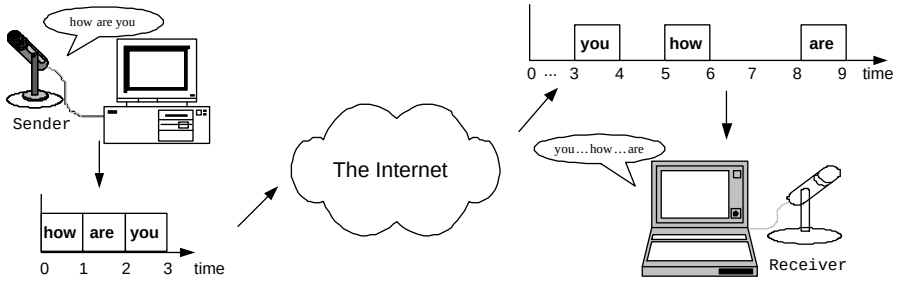


Figure 3.1: Effect of jitter and out-of-sequence in voice transmission.

in delivering the email is not significant. However, real-time delivery services are delay sensitive. If the information they carry cannot be delivered to the destination within a time limit, either the information becomes no longer useful to the receiver or the service quality drops dramatically.

There are two issues in real-time delivery when the sender transmits a continuous stream of voice packets over the Internet. Both issues are highlighted in Figure 3.1. In Figure 3.1, "how-are-you", in a consecutive sequence order, labelled with *1*, *2* and *3* for each word in the packets. The first issue is related to the order in which packets arrive at their destination. The three packets arrive at the destination in a different order from the original one becoming "you-how-are" in this case. The second issue is jitter, which means the delay between successive voice packets differs significantly in the destination. The continuous stream of voice is fragmented and the playback does not occur at correct time compared to the original voice packets.

In conclusion, there are two aspects of real-time delivery that protocol software is required to handle:

- **Order** Data in an out-of-order sequence in the receiver must be processed in the same order in which they were sent before playback.
- **Time** The receiver must obtain time information from the sender to know at what time the data in the packet should be played back.

It has been argued above that there are two problems for real-time delivery. In the next sections, it will describe how Real-Time Transport Protocol (RTP) is used to handle these two problems.

3.1.1 Real-Time Transport Protocol (RTP)

The RTP protocol is designed to support real-time delivery and to handle the out-of-order problem and timing problem. To cope with these problems, the protocol assigns a sequence number and a timestamp to the packet header. In Figure 3.2, the sequence number is denoted by SN, the timestamp is denoted by T. The sequence number enables the receiver to process the packets in the same order as they were sent.

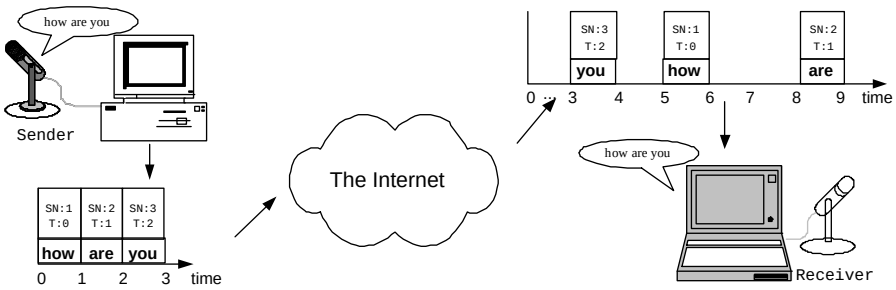


Figure 3.2: The RTP protocol handles jitter and out-of-sequence in voice transmission.

The sequence number also allows the receiver to detect packet losses. Once ordered, the original timing relationship of the real-time data can be recovered by reading the timestamp. In the case of encoded audio, timestamps inform the receiver when to play the audio through the speaker. Therefore, delay jitter is compensated. To monitor the data delivery and provide minimal control and identification functionality, the Real-Time Transport Control Protocol (RTCP) is utilized in conjunction with RTP. The discussion of RTCP packet format will be given in Section 3.1.2.

Before discussing the RTP packet format, some definitions, taken from [19] are presented as follows:

- RTP payload: The data transported by RTP in a packet, for example, audio samples or compressed video data.
- RTP packet: A data packet consisting of the fixed RTP header, a possible list of Contributing source (CSRC) identifiers, a possible RTP header extension and RTP payload.
- RTCP packet: A control packet consisting a fixed header part similar to that of RTP data packets, followed by structured elements that vary depending upon the RTCP packet type.
- Port: The abstraction that transport protocols use to distinguish among multiple destinations within a given host computer.
- Transport address: The combination of a network address and port that identifies a transport-level endpoint, for example, an IP address and a UDP port.
- RTP session: The association among a set of participants communicating with RTP. For each participant, the session is defined by a particular pair of destination transport address (one network address plus a port pair for RTP and RTCP).
- End system: An application that generates the content to be sent in RTP packets and/or consumes the content of received RTP packets.

Figure 3.3 depicts the structure of an RTP packet. The first twelve bytes must be presented in every RTP packet, while the CSRC list and RTP header extension are

optional. Specific details regarding the use of these header fields, RTP and its profiles are given in [19] and [20]. A short description of each RTP packet field is described as follows:

- **Version** (V: 2 bits) identifies the version of RTP. The current version is *2*.
- **Padding** (P: 1 bit) identifies whether the packet contains one or more additional padding. For example, if P is *0*, no padding is performed.
- **Extension** (X: 1 bit) identifies whether the fixed header is followed by exactly one header extension. For example, if X is *1*, a header extension is added to the fixed RTP header.
- **CSRC count** (CC: 4 bits) contains the number of CSRC identifiers that follow the fixed header. Since CC is four bits, the maximum number of CSRC is *15*.
- **Marker** (M: 1 bit) is used to allow significant events such as frame boundaries to be marked in the packet stream. For example, during a voice conversation, first packet is distinguished by setting the marker bit into *1* and other packets are set to zero.
- **Payload type** (PT: 7 bits) specifies the RTP payload type. For example, one audio encoding (PCM) uses a PT as *0*.
- **Sequence number** (16 bits) denotes the sequence in which each RTP packet is sent. This is done by incrementing the sequence number by one while sending subsequent packets. The sequence number may be used by the receiver to detect packet loss and to restore packet sequence. The initial value of the sequence number is chosen at random, which increases security by making it difficult for attackers to guess the sequence information.
- **Timestamp** (32 bits) denotes the time at which the first byte of data in the packet was sampled. More importantly, a sender is required to increment its timestamp clock continuously, even if no signal is detected and no data is sent. Therefore, a sender does not have to generate useless packets when no data need to be sent. A receiver can determine from the timestamp how long the gap is. RTP specifies that the initial timestamp must be chosen at random, as for the sequence number.
- **Synchronization source** (SSRC) identifier (32 bits) identifies the source of an RTP packets stream. All packets from a synchronization source belong to the same timing and sequence number space allowing a receiver to group packets by synchronization source for playback. Examples of synchronization sources include the sender of a packet stream derived from a signal source such as a microphone or a camera. The SSRC identifier is chosen randomly, with the intent that no two synchronization source within the same RTP session will have the same SSRC identifier.
- **CSRC list** (0 to 15 items, 32 bits each) identifies the contributing sources for the payload contained in the packet. It is produced by an RTP mixer, an intermediate

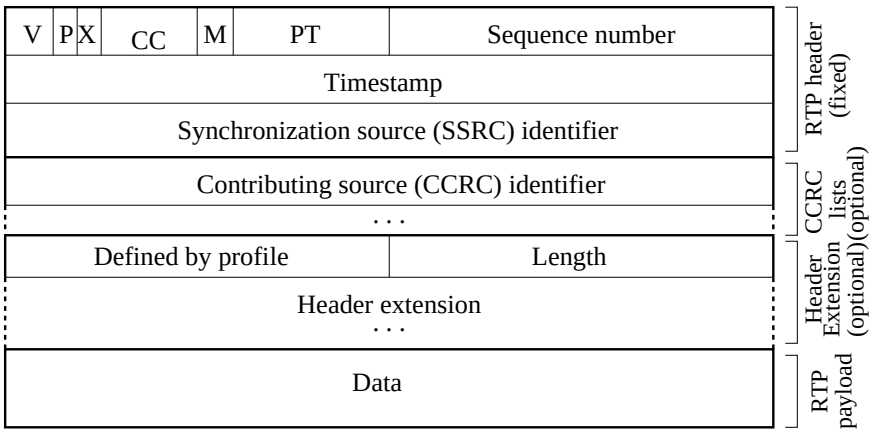


Figure 3.3: The format of an RTP packet.

system that receives RTP packets from one or more sources. The mixer inserts a list of the SSRC identifiers of the sources that contributed to the generation of a particular packet into the RTP header of that RTP packet. The number of identifiers is given by the CC field. If there are more than *15* contributing sources, only *15* can be identified.

- **RTP header extension** (variable length) is provided to allow individual implementations to experiment with new functions that require additional information to be carried in the RTP header.

Although RTP provides information that a receiver needs to recreate real-time output, the RTP header does not contain any information for endpoints to monitor the quality of data delivery.

3.1.2 RTP Control Protocol (RTCP)

The RTCP protocol is designed for the purpose of providing the feedback to the sender and monitoring the quality of data delivery by working in conjunction with RTP. Considering the following example. The sender has access to a broad-band network and a receiver only has access to a narrow-band network. At the beginning of the transmission, data packets transferred from the sender have been delayed at the receiver. Before the data packets are played out, new packets have been coming into the receiver. Hence, it causes link congestion and packet loss. However, it would be avoided if the receiver could provide timely feedbacks to the sender about its network situation and current packet loss by utilizing the RTCP protocol so that the sender can adjust the packet delivery speed.

The RTCP protocol is used to transmit periodically control packets to all participants in the session. The primary control packets for RTCP are the RTCP Sender and Receiver Report (SR and RR) [19], which are two basic RTCP packet types. Each RTCP packet begins with an 8-byte fixed header similar to that of RTP packet, followed by structured

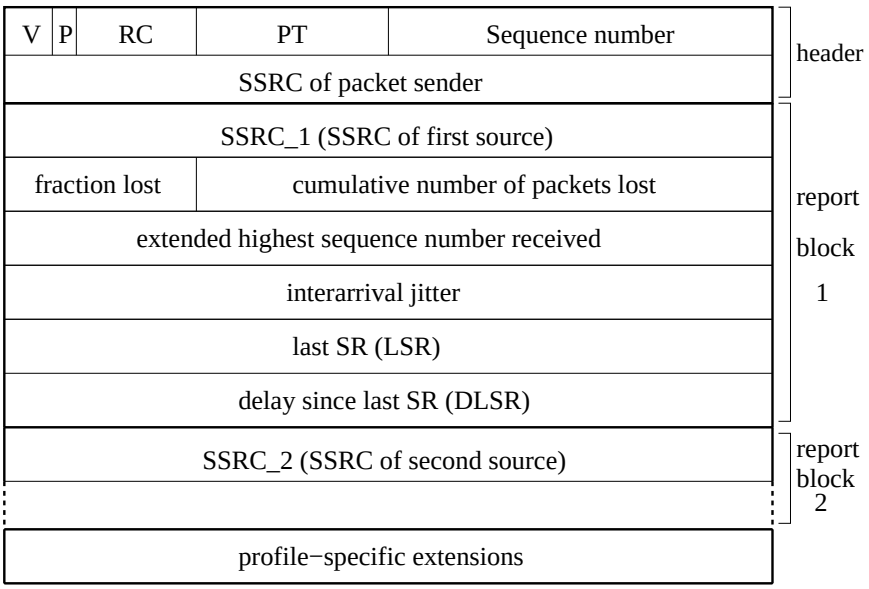


Figure 3.4: The format of an RTCP Receiver Report (RR) packet.

elements that are determined by the type field in the header.

Figure 3.4 depicts the format of an RTCP Receiver Report (RR) packet. It mainly consists of two sections, possibly followed by a third profile-specific extension section if defined. The first section (header) is 8-byte long. The second section contains zero or more reception report blocks depending on the number of other sources heard by this sender. Each reception report block conveys statistics on the reception of RTP packets from a single synchronization source. The fields in the RTCP RR packet are briefly discussed as follows:

- **Version** (V: 2 bits) identifies the version of RTP. The current value is *2*.
- **Padding** (P: 1 bit) identifies if the packet contains one or more additional padding.
- **Reception report count** (RC: 5 bits) is the number of reception report blocks contained in the packet.
- **Packet Type** (PT: 8 bits) contains the constant 201 to identify this as an RTCP RR packet.
- **Length** (16 bits) is the length of the RTCP packet in 32-bit words minus one, including the header and any padding.
- **SSRC of sender** (32 bits) is the SSRC identifier for the originator of the RR packet.
- **SSRC_n** (32 bits) is the SSRC identifier of the source to which the information in the RR block pertains. Here, n can be *0*, *1* until the number defined in the RC field.

- **Fraction lost** (8 bits) is the fraction of RTP packets from source SSRC_n lost since the previous of RR packet was sent. This fraction is defined to be the number of packets lost divided by the number of packets expected.

- **Cumulative number of packets lost** (24 bits) is the total number of RTP packets from source SSRC_n that have been lost since the beginning of reception. This number is defined to be the number of packets expected less the number of packets actually received, where the number of packets received includes any which are late or duplicates.

- **Extended highest sequence number received** (32 bits) extends the sequence number from SSRC_n with the count of sequence number cycles. The least significant *16* bits contain the highest sequence number received in an RTP packet from source SSRC_n, and the most significant *16* bits contain the count of sequence number cycles. The idea behind this is as follows. As mentioned above, the field of the sequence number in an RTP packet is *16* bits, hence the maximum of sequence number is *65535*. If the sender would send the *65536th* RTP packet or more, the sequence number in the header of RTP packet will start counting from *0* in another cycle.

- **Interarrival jitter** (32 bits) is an estimate of the statistical variance of the RTP packet interarrival time, measured in timestamp units and expressed as an unsigned integer.

- **Last SR timestamp** (LSR: 32 bits) is the middle *32* bits out of *64* in the Network Time Protocol (NTP) timestamp received as part of the most recent RTCP SR packet from source SSRC_n.

- **Delay since last SR** (DLSR: 32 bits) is the delay expressed in units of 1/65536 seconds, between receiving the last SR packet from source SSRC_n and sending this RR block.

Receiver reports are important for two reasons. First, they allow all receivers participating in a session as well as a sender to exchange information on reception conditions of other receivers. Second, they allow receivers to adapt their reporting rates to avoid using excessive bandwidth and overwhelming the sender. It is suggested that the fraction of the bandwidth allocated to the RTCP traffic calculated by the bandwidth for all RTCP traffic over the overall bandwidth for this session be fixed at 5% and that receiver reports generate less than 75% of total RTCP traffic [19].

3.2 Benchmarking RTP/RTCP Processing

The RTP/RTCP protocols have been used for delivering real-time (multimedia) data over the IP network since they provides functionalities suited for carrying real-time data, e.g., timestamps and control mechanisms for synchronizing different streams with timing properties. Therefore, an RTP/RTCP benchmark plays an important role in

investigating real-time network processing in the higher layer .

The RTP protocol is responsible for sending and receiving RTP packets, and the RTCP protocol is responsible for providing control information for RTCP packets. Therefore, the benchmarking suite for RTP and RTCP consists of an RTP Sender benchmark, an RTP Receiver benchmark and an RTCP Processing benchmark. The implementation of the benchmarks are based on existing software codes [10], [5] and [19]. We use the methodology described in Chapter 2 for this benchmarking suite. Detailed implementations on the benchmarks are specified from three aspects: function, measurement and environment. First, the function aspect specifies the core algorithms used in the benchmarks, possible functions to be realized in each benchmark, and the manner to implement these functions. Second, the measurement aspect specifies the metrics for evaluating performance of the benchmarks and the key architectural characteristics of the benchmarks. The measurement aspect also verifies the correctness of the benchmarks. Third, the environment aspect specifies simulation environment and the interface (input/output) of the benchmarks. In all these three specifications, measurement specification and environment specification on simulation environment are the same for all created benchmarks in this thesis.

This section is organized as follows. Section 3.2.1 specifies the function and interface environment aspects of the RTP Sender benchmark. Section 3.2.2 specifies the function and interface environment aspects of the RTP Receiver benchmark. Section 3.2.3 specifies the function and interface environment aspects of the RTCP Processing benchmark. Section 3.2.4 presents the common measurement and simulation environment aspects for all benchmarks.

3.2.1 RTP Sender

After session connection[1] between the end systems, every particular packetization interval mediadata, e.g., 20ms recommended by RFC 1889 [19], can be sent out after it is encapsulated into an RTP packet with preceded by an RTP header.

Function specification: Three functions are included in the RTP Sender benchmark, reading data from input, generating RTP packets and writing the RTP packets to the output.

- **Reading data from input** is to obtain a stream of voice data as an RTP payload and to allocate an buffer to the stream. The PCMU encoded audio data stream are read from a file and taken as the RTP payload. The sampling rate for the audio data is *8000HZ* and PCMU specifies audio data are encoded as *8* bits per sample. If the time interval for the audio data is *20* ms, the input buffer is set into *160* bytes, which derives from the following equation:

$$20ms \div 10^3 \times 8000(persample/s) \times 8bits/persample = 1280bits$$

[1] Session connection is out of discussion in this thesis.

$$1280bits = 1280bits \div 8bits/perbyte = 160bytes$$

- **Generating an RTP packet** consists of building an RTP header, prepending the RTP header to the RTP payload to be an RTP packet, and increasing the timestamp and sequence number for the next RTP packet.

  Building an RTP header possibly consists of building three parts: a fixed RTP header, a possible RTP CSRC lists and a possible RTP extension. First, in the fixed RTP header, the initial values of sequence number, timestamp and SSRC identifier are generated randomly by using the algorithm defined in [19]; The field of PT is set into 0 since PCMU audio data is used [20]; The field of M for the first RTP packet is set to 1 to indicate the beginning of the voice conversation, and the M for other RTP packets are set into 0; The field of CC is set into 0 if the CSRC lists are $NULL$; Otherwise, it is set into the number of the CSRC lists if it is less than 15 or set into 15 if it is larger than 15. Second, building an RTP CSRC lists means coping the CSRC lists in the memory to the RTP header structure. Once the session is set up, each participant holds a table where the SSRC identifiers for all participants the CSRC identifiers for all contributors are stored. Therefore, if there is no information for CSRC in the memory, the CSRC lists in the RTP packet are empty. Third, building the RTP extension means coping the extension information in the memory to the RTP packet structure if the information is available. If the information is not available in the memory, the header extension is not necessary to generate. This is similar with the CSRC lists.

  Each RTP packet has its own RTP header, some fields of which can be the same with the one of other packets, some not, Such as sequence number, and timestamp. As mentioned before, the sequence number increments by one for each RTP packet. While timestamp computation is different for different payload types. For audio, the timestamp is incremented by the packetization interval times the sampling rate. In our case that audio packets contain 20ms of audio sampled at 8000HZ, the timestamp for each block of audio is increased by *160*, even if the block is not sent due to silence suppression. For video, the timestamps generated depend on frame rate, which is out of discussion in this thesis.

- **Writing the RTP packet** is done in two ways, writing packets to the memory or to the hard disk. Writing packets to the hard disk is necessary because the written file will be used as the input of the RTP Receiver benchmark.

  When writing to the memory, an enough memory space should be allocated. The length of an RTP packet is not fixed since the length for the header of the RTP packet is not fixed. Therefore, a long enough buffer for saving the RTP packet has to be allocated. It would be not necessary for generating an RTP header if the buffer length is not long enough to save the RTP packet. Therefore, first calculation about the true length of the RTP packet and comparison it with the buffer length are processed.

  When writing to the hard disk, two issues have to be considered. First, in order to be convenient to read RTP packets from the hard disk, the length of each RTP

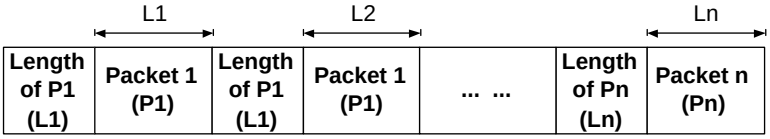


Figure 3.5: The structure used to write RTP packets to the hard disk.

packet has to be provided before reading the RTP packet. Figure 3.5[2] depicts the structure used to store RTP packet. Before $P1$ is written to the hard disk, $L1$ has to be written first. So do all other RTP packets. Second, in network traffic the sequence number of an RTP packet, from $P1$ to Pn, may be in order or out of order. In-order sequence gives the correct order of $P1$ to Pn incremented by one. While out-of-order sequence gives incorrect order where the sequence number of $P1$ may be larger than the sequence number of, for example, $P2$ or $P3$, which represents packet delay. In-order sequence writing is straightforward. Whenever an RTP packet is generated, it is immediately written to the hard disk, because generating an RTP packet ensures that the sequence number for each RTP packet increases by one. Out-of-order sequence writing is more complex. The difference between sequence number of two neighboring RTP packets can be relatively small, say, 5, or considerably large, say, 40, which represents a short delay or a long delay, respectively. Writing out-of-order packets uses the following algorithm. First a buffer is defined into an array of N elements, each element of which consists of the structure depicted in Figure 3.5. Every generated RTP packet together with its length are stored into the buffer until the buffer is full or the packet generation is finished. In the first case that the buffer is full, the i-th element and the $(N/2+i)$-th element[3] are written to the hard disk in that order until the writing loop is finished. In the second case that the packet generation is finished while the buffer is not full, the number of elements stored in the buffer is calculated first, and then written to the hard disk.

Environment specification - interfaces: The RTP Sender benchmark takes an audio file as input, the size of which is *4MB*. Every *20 ms*, that is, *160 bytes* voice data are read and used to generate RTP packets until the end of the file. Therefore, the input buffer is set into *160 bytes* for *20 ms* voice. Since it is not necessary for RTP to have a fixed packetization interval, in another words, *20 ms* is not fixed number for voice interval, implementation can choose any reasonable value. In that case, input buffer needs to be set into its corresponding size. Every generated RTP packet is written to the memory or to the hard disk as output. The output buffer is set into *10000 bytes* for an RTP packet.

3.2.2 RTP Receiver

Comparing with the RTP Sender benchmark, the RTP Receiver benchmark is more complex. It has to handle the main problems for real-time delivery: out-of-sequence

[2] $L1$ denotes the length of the first packet ($P1$), $L2$ denotes the length of the second packet ($P2$), and so on.

[3] N denotes the maximum elements in the array and i is increased from 0 till $N/2-1$ by 1.

and jitter. In addition, it has to update statistic information which will be used for generating RTCP RR packets.

Function specification: The RTP Receiver benchmark consists of four functions: an input buffer management, data parsing, statistics updating and queue management, which are discussed as follows.

- **Input buffer management** Incoming packets must be placed into the buffer for further processing. The input buffer is not designed to accommodate one RTP packet, but about *200* packets[4], which can hold voice data for about 4 seconds. The reason behind this is that every packet is not played out immediately when it is coming. In order to manage the buffer conveniently, a pointer is arranged to point the place where a packet starts to be saved. When a packet is coming, the function first checks if the left space of the buffer is big enough for the packet. If the space is sufficient, the packet will be put into the buffer followed with the last packet and the pointer is arranged to the next position of the end of the packet for the next packet. If the space is not sufficient, the packet is located to the beginning of the buffer. Therefore, it will be happened that a new packet is put into the beginning of the buffer while the original packet in that position is not played out. In this case, the original packet is taken as discarded.

- **Data parsing** A packet is received as a format of a string which represents an RTP packet. As mentioned before, an RTP packet is composed of an RTP fixed header, a possible CSRC list, a possible RTP header extension and an RTP payload. Therefore, the string has to be converted into the *rtp_packet* structure (Figure 3.3) defined for an RTP packet. The *rtp_packet* structure provides access to every field of the RTP packet and has four parts. The first is a pointer to a fixed RTP header structure and a CSRC list with a variable length. The second is a pointer to an RTP header extension structure. The third is a pointer to an RTP payload. The final part is the length of the RTP payload.

  The function of the data parsing does this conversion. It takes a pointer to an individual string saved in the input buffer, allocates a pointer to a right place where the RTP header, the possible RTP extension and the RTP payload are, calculates the right length of the RTP payload and returns a pointer to the *rtp_packet* structure. In the conversion, two issues should be considered. First, *16*-bit and *32*-bit quantities must be converted from network byte order to host byte order for further processing, e.g., the sequence number field in the RTP header. Second, it makes the pointer to an RTP payload complex that an RTP header extension is not presented in every RTP packets and the length of CSRC lists is variable. The length of CSRC lists is calculated by multiplying the *CC* field in the RTP header by four. A possible RTP header extension is checked by the *X* field in the RTP header to ensure if it is presented. If *X* is equal to 1, it is presented. Then, the 16-bit length field in the RTP header extension also has to be converted from

[4]The number of RTP packets in the input buffer can not be fixed since the length for each coming RTP packet is not fixed.

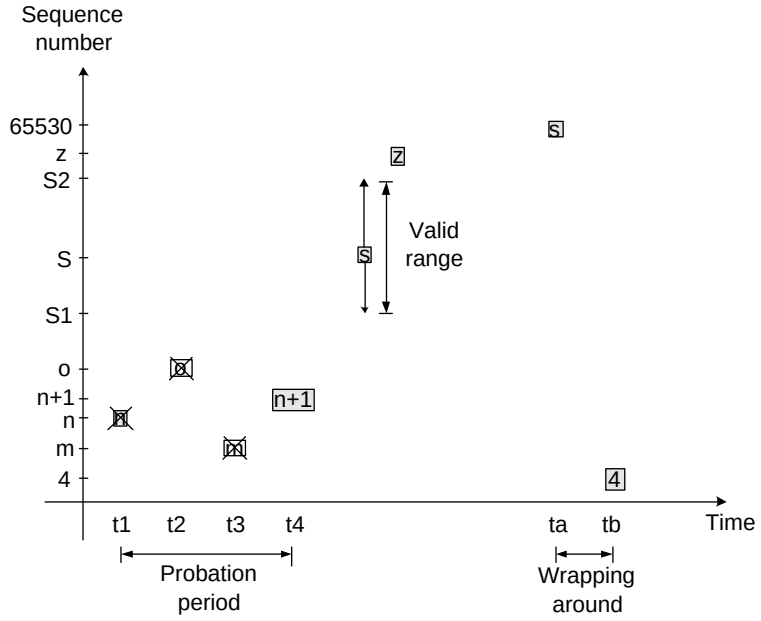


Figure 3.6: Issues for sequence number processing.

network byte order to host byte order. Otherwise, the pointer to the RTP payload could be allocated to the wrong place.

Besides the conversion, the function of the data parsing also verifies every received packet as an RTP packet. It is checked by two criteria. The first is that the RTP version field must equal to 2. The second is that the payload type of the RTP packet must equal to 0 since the PCMU encoded voice data is used as a source. A packet that can not meet the requirements is not precessed further and a new iteration to receive a packet from the input is begun.

- **Statistics updating** As mentioned above, RTCP performs all statistics collection and generates a receiver report periodically. To do so, RTCP needs information about the most recently received packet and the jitter. The function of statistics updating is designed for this purpose. It consists of two aspects: processing the sequence number of each coming RTP packet and estimating the interarrival jitter.

  The RTP receiver analyzes the sequence number of each coming RTP packet and checks its validity. Three issues have to be considered for the implementation (depicted in Figure 3.6).

  – When an RTP receiver receives an RTP packet from a sender for the first time, the receiver takes the packet as invalidity and ignores it since the receiver does not have the information of the SSRC identifier for the sender. If the sender continues sending RTP packets to the receiver, the receiver should not ignore all, otherwise, the connection can not be set up. Therefore, a probation period is designed to ensure that the sender is not valid until $MIN\_SEQUENTIAL$,

say 2, packets with sequential sequence numbers have been received. The initial probation variable is set into $MIN\_SEQUENTIAL$. When the first packet with sequence number n is received, the number n is saved as a current maximum value of the sequence number received, the probation variable is decreased by 1, and a state of a bad packet is returned by the function. When a packet with sequence o or m is received, the packet is discarded and a state of a bad packet is returned because either m or o is not the next number of n. When a packet with sequence number $n + 1$, the packet is taken as the first valid packet, the current maximum value of the sequence number is set into $n + 1$, and the probation period ends. The validity check can be made stronger requiring more than two packets in sequence. The disadvantages are that a larger number of initial packets will be discarded and that high packet loss rates could prevent validation.

- This function must handle the case where the sequence numbers wrap around. The idea behind this is because the number of bits of the sequence number field in an RTP header is 16. Therefore the largest value for the sequence number is 65535. If a receiver receives a last RTP packet with the sequence number 65535, the current RTP packet with the sequence number 0 received should be a valid packet with wrapping around once. To do this, the function first checks to determine whether the sequence number in the coming packet is within a fixed distance of the expected sequence ([19] suggests that the sequence number is considered valid if it is no more than $MAX\_DROPOUT$ ahead of the current maximum sequence number nor more than $MAX\_MISORDER$ behind.). If the new sequence number is valid and it is less than the current maximum sequence number, wrapping around happens and the count of sequence number cycles is incremented by 1. In Figure 3.6, if the current maximum sequence number is s, the sequence number within $s1$ and $s2$ are considered valid where $s1$ is equal to $(s - MAX\_MISORDER)$ and $s2$ is equal to $(s + MAX\_DROPOUT)$. If the current maximum sequence number is 65530, the new sequence number with 4 is coming. Because it is ahead of 65530 modulo 65536 and it is smaller than 65530, it is valid and the function wraps around by 1.
- The function must handle the issue that the sequence number makes a so large jump that the sequence number is out of valid range. Suppose that the current maximum sequence number is s and a packet with the sequence number z is coming, the packet is discarded because it is out of valid range of the sequence s. If another new packet with the sequence number x is coming, two issues rises because x can be in the valid range of s or not. For the first case, x is taken as normal. For the second case, if x is equal to $z + 1$, the function takes it as the first packet, like the probation period; if x is not equal to $z + 1$, it will be discarded and a state of a bad packet is returned.

Besides the sequence processing in the function of statistics updating, the function also estimates the interarrival jitter which RTCP needs to generate a receiver report. The interarrival jitter is defined to be the mean deviation of the difference

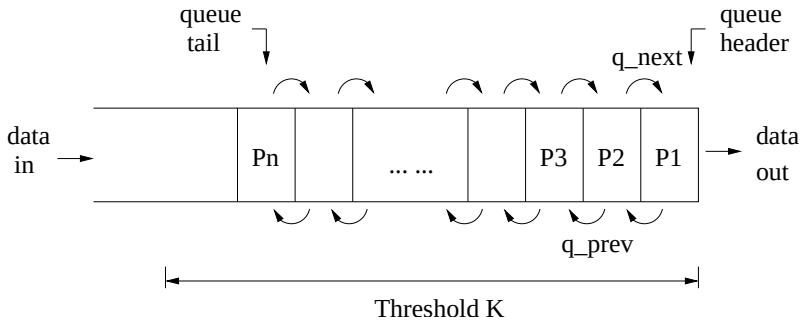


Figure 3.7: The queue structure.

D in packet spacing at the receiver compared to the sender for a pair of packets ([19]). Therefore, two steps need to recursively calculate the interarrival jitter: estimation the jitter for a packet and calculation the mean of the jitter. The first step is calculated by the following equation:

$$D_i = (R_i - R_{i-1}) - (S_i - S_{i-1}) \tag{3.1}$$

Where D_i denotes jitter estimation for the ith packet, R_i and R_{i-1} denote the time of arrival in RTP timestamp units for the ith and $(i-1)$th packet, respectively. S_i and S_{i-1} denote RTP timestamp of the ith and $(i-1)$th packet, respectively. The second step is calculated by the following equation:

$$J_i = 15/16 J_{i-1} + 1/16|D_i| \tag{3.2}$$

Where J_i denotes temporal average of the jitter for the ith packet, J_{i-1} denotes for the $(i-1)$th packet.

- **Queue management** Because real-time data is sensitive to delay and jitter is a common occurrence for real-time delivery, playback can not start immediately when data first arrives. Otherwise, the playback will be interrupted when the next packet has not arrived while the last packet is already played out. Instead, incoming data is ordered by its sequence number and then placed in a buffer, known as the *jitter buffer*. Figure 3.7 depicts the structure of the queue. It is a first-in-first-out queue with a threshold K. When the queue is full, the packets stored in the queue is played out. Two pointers point to the header and the tail of the queue respectively. Every packet in the queue, say $P2$, also has two pointers, q_next and q_prev, which point to the packet $P1$ and $P3$ respectively. The function of queue management is to arrange data moving into the queue or out of the queue. It consists of two procedures [5]: one is to insert every incoming packet into the queue by the order of the sequence number, and the other is to dequeue the packet out of the queue.

  Dequeuing is simple. Since it is a first-in-first-out queue, the packet always moves out from the header of the queue if the header of the queue is not $NULL$. Once the packet is moved out, the pointer to the header of the queue should be reallocated

to the new position, and the *q_next* pointer of the following packet should be set into $NULL$. For example in the Figure 3.7, when $P1$, where the header of the queue is pointing, is dequeued, the header of the queue will be arranged to point to $P2$, and the *q_next* pointer of $P2$ is set into $NULL$.

Inserting packets to the queue is more complex than dequeuing because packets can arrive out of order and packets need to be inserted in the order of the sequence number. Thus, when a packet is coming in, comparison its sequence number with sequence numbers of all other packets in the queue is precessed iteratively from the tail of the queue, a location is arranged where the packet should be inserted and the pointers for the queue is adjusted to the right location where they should pointed to. Three possible locations can be arranged for the packet based on its sequence number. First, it is arranged as a new tail of the queue if its sequence number is larger than the current tail of the queue. The tail of the queue is reallocated into the new packet. The pointer *q_prev* of the original tail is pointed to the new tail, the pointer *q_next* of the new packet is pointed to the original tail, and the pointer *q_prev* of the new packet is set into $NULL$. Second, Moving the comparison in the direction of from the tail to the header continues if it is not larger than the sequence number of a packet in the queue. It will be arranged somewhere in the middle of the queue if the moving is stopped before the header. Suppose that the sequence number of its left packet is larger than the one of its right packet, pointers adjustment is done as follows: The *q_prev* pointer and *q_next* pointer of the packet is arranged to point to its left and right neighboring packets respectively and the pointer *q_next* of its left packet and the pointer *q_prev* of its right packet is arranged to point the packet. Third, it can be arranged before the header of the queue as the new header if its sequence number is less than the one of the header. The header of the queue is reallocated into the new packet. The *q_next* pointer of the original header is pointed to this packet, its *q_prev* pointer is arranged to point the original header and its *q_next* pointer is set into $NULL$.

Environment specification - interfaces: The RTP Receiver benchmark takes RTP packets stored in a file as the input. The file is the output of the RTP Sender benchmark, where every RTP packet is stored as the fixed structure depicted in Figure 3.5. To obtain an RTP packet, two steps are done: first reading a 4-byte stream as a length of a packet, then reading a stream by the length as an RTP packet. The RTP packet is stored in the input buffer for further processing, the size of which is set into *64KB*. The output of the benchmark is stored in two places for two different purposes respectively, one is written to the memory for simulation, the other is written to a file for checking the correctness of the functions in the benchmark. If the file is the same with the file used as the input of the RTP Sender benchmark, it means the functions are correct because final result of the RTP Receiver benchmark is to obtain the same voice signal as the one in the sender.

3.2.3 RTCP Processing

The RTCP protocol mainly focuses on providing RTP control packets, one of which is the RR packet (Figure 3.4). It contains reception quality reporting, used by senders

to adapt their transmission rates or encodings dynamically during a session. However, if the RR packets from each participant are sent at a constant rate, the control traffic would grow linearly with the number of the participants. Therefore, the rate must to scaled down. RTCP has to adjust the interval between reports to achieve scalability to large group sizes.

Function specification: In RTCP Processing benchmark, two functions are investigated: building the RTCP RR packet and computing the RTCP transmission interval.

- **Building the RTCP RR packet** consists of two aspects, generating a RTCP header and building the report block. First, Generating an RTCP header is similar with generating an RTP header in the RTP Sender benchmark. In the RTCP header, the V field is set into current value, 2 and the PT field is set into a fixed number, 201. The P is set into 1 if this RTCP packet contains some additional padding, or it is set into 0 if there is no padding in this RTCP packet. The RC filed is set into the number of the report block contained in this RTCP packet. The length field is calculated by the following equation: $length = 1 + S \times N/4$, where S denotes the size of the report block, N denotes the number of the report block, 4 denotes the length is measured in the unit of 32-bit words. Second, a report block consists of a number of fields (depicted in Figure 3.4), which are determined by different computation [19].

  - The *extended highest sequence number received* (es_max) field is calculated by the equation: $es\_max = (cycle >> 16)|max\_seq$, where $cycle$ denotes the number of wrapping around in the sequence number, and max_seq denotes the current maximum sequence number received.
  - The *cumulative number of packets lost* field is defined to be the number of packets expected ($n\_expected$) less the number of packets actually received ($n\_received$), where $n\_expected$ can be calculated by the difference between the highest sequence number, es_max, and the first sequence number received, and $n\_received$ is simply the count of packets as they arrive.
  - The *fraction lost*(FL) field is calculated from the difference between the $n\_expected$ and $n\_received$ across the interval.

    $$lost\_interval = (n\_expected_i - n\_expected_{i-1}) - (n\_received_i - n\_received_{i-1})$$

    $$FL = \frac{lost\_interval << 8}{n\_expected_i - n\_expected_{i-1}}$$

    Where $n\_expected_i$ and $n\_expected_{i-1}$ denote the number of packets expected when ith and the previous, that is, $(i-1)$th, reception report is generated, and $n\_received_i$ and $n\_received_{i-1}$ denote the number of packets actually received when ith and the previous, that is, $(i-1)$th, reception report is generated.
  - The *interarrival jitter* field is calculated by Equation 3.2.

- The *LSR* field is calculated from the NTP timestamp received in the sender report from the senders. The full resolution NTP timestamp is a 64-bit unsigned fixed-point number with the integer part in the first 32 bits and the fraction part in the last 32 bits. The LSR is the middle 32 bits out of it, that is, the low 16 bits of integer part and the high 16 bits of fraction part.
- The *DLSR* field is calculated by the time difference between receiving the last sender report from the senders and sending the reception report block. It is measured in units of 1/65536 seconds. Because time in the system is stored as the structure which consists of two elements: tv_sec representing as the number of seconds, tv_usec representing as the number of microseconds. This time structure is changed into corresponding value as an integer by:

  $$t.tv\_sec \times clock + ((int)((double)t.tv\_usec \times .000001 \times (double)clock))$$

  Where t is a time with the time structure, and *clock* is the clock rate, for example, 65536 for the DLSR calculation.

- **Computing the RTCP transmission interval for scalability** is calculated by the following equation:

  $$T_d = max(T_{min}, CL(t)) \tag{3.3}$$

  Where T_{min} is $2.5s$ for the initial packet, and $5s$ for all other packets, C is the average RTCP packet size divided by 5% of the session bandwidth, $L(t)$ represents the number of users within a multicast group that have been heard from at time t, and the initial value at time 0, $L(0) = 1$ when the user joins the group. To avoid traffic bursts from unintended synchronization with other sites, the actual interval is then computed as a random number uniformly distributed between 0.5 and 1.5 times T_d. Before the calculation T_d, the average RTCP packet has to be updated by the size of the report packet just sent and the number of users has to be estimated at each time when the report is sent.

Environment specification - interfaces: The RTCP Processing benchmark takes the same file as the RTP Receiver benchmark as the input. The input buffer for RTCP packet is set into 100 bytes which can contain one report block in an RTCP RR packet. the output of the benchmark, every generated RTCP packet, is stored in the memory. The function of the RTCP transmission interval computation is called after building an RTCP to calculate the delay until the next should be built.

3.2.4 Measurement and Simulation Environment

This section specifies the common aspects for the created benchmarks mentioned above on measurement and simulation environment.

Measurement specification: It describes three aspects in measurement. First, *verification of benchmarks* has to be checked before measuring the performance of the benchmarks. The correctness of the benchmarks are verifies by comparison the input of the

RTP Sender benchmark and the output of the RTP Receiver benchmark. If there is no difference between the input and the output, the correctness of the benchmarks are ensured since the final results at the receiver are to obtain the same voice signal with the one at the sender. Second, *the performance metric* for evaluating the performance of all benchmarks is the number of clock cycles, which captures how fast the benchmarks are executed and which functions in the benchmark are time-critical. For the time-critical functions, it could be implemented in specialized hardware to obtain the performance gain. Third, *the architectural characteristics* for the benchmarks are also investigated in order to understand the features of the benchmarks. These characteristics are as follows:

- Instruction Level Parallelism (ILP) is measured by instruction per cycle (IPC) which shows us data-level parallelism and dependency of the instructions. IPC value is high when the dependency of the instruction with a program is low. A high IPC means that it can be and should be exploited in the processor design because parallelism is one of the essential performance enhancing techniques for processing design.
- Branch Prediction Accuracy is measured by branch address-prediction rate (APR) and branch direction-prediction rate (DPR). A high branch prediction accuracy means that less exotic/complex branch predictors can be utilized, e.g., to keep the pipelined field with useful instructions.
- Instruction distribution is measured by the frequency of load instructions, store instructions and branch instructions, which is important in determining the features of the benchmarks. Depending on the instruction distribution, design decisions can be made whether to include certain functional units or not.
- Cache behavior is measured by the number of cache accesses and miss ratios. Both data cache and instruction cache are simulated. dl1 stands for first level data cache, il1 stands for first level instruction cache and il2 stands for second level instruction. The reason for the investigation on cache behavior is that cache has a profound effect on performance.

The four architectural characteristics on the created benchmarks mentioned above are compared with the applications from NetBench and MediaBench to understand the features of the benchmarks and to highlight the difference between real-time network processing with other processing.

Environment specification - simulation environment: The *sim-outorder* simulator from the SimpleScalar tool set (Version 3.0) is utilized for all benchmarks. This simulator allows to measure the performance of the benchmarks and to determine the time (in clock cycles) spent in each function in relation to the total time of each benchmark. The measurement is performed by utilizing instruction annotations in the *sim-outorder* simulator. We have introduced a NOP instruction to signify the start of a function and another NOP to signify the end of the function. More specifically, the overall simulation clock cycle (called *sim_cycle*) is noted in both cases. By subtracting the starting sim_cycle from the ending sim_cycle, the precise number of clock cycles

spent in executing a given function can be calculated.

Both the benchmark and the simulator need to be modified to support the instruction annotations and to calculate accurate clock cycles. In the benchmarks, an same annotation is added before calling the function and after executing the function. In the simulator, the sim_cycle value has to be recorded once the annotation is read. Therefore, the sim_cycle is recorded twice, one is recorded as $cycle1$ before calling the function, the other is as $cycle2$ after executing the function. Thus, the substraction $cycle1$ from $cycle2$ is the number of clock cycles for the function execution. Two more points should be paid attention to during the modification of the benchmark and the simulator:

- In the benchmark, if a function needs to be simulated, the annotation should be added where the function is called, not inside the function. It is important especially when the function has different return statements which could be existed in the branch of the function. Otherwise, the simulator would not calculate the accurate clock cycles for the function.
- The modification for the simulator mentioned above is only to calculate the clock cycles for a function when the function is called once. If a function is called more than once, and the total cycles for execution the function is required to generate, the modification can be done as follows. When the simulator identifies the annotation at an odd time, the clock cycles are recorded as $cycle1$; When the simulator identifies the annotation at the next a even time, that is, an odd number plus 1, the clock cycles are recorded as $cycle2$. Final cycles for the function is the summation of every difference between $cycle1$ and $cycle2$.

The simulation environment is set to all created benchmarks in this thesis. Once the clock cycles for each function in a benchmark is known, we can perform profiling to find time-critical functions in the benchmark.

3.3 Conclusions

This chapter reviewed the real-time delivery over the IP network, which has become a popular service in recent years. The characteristics of this service was introduced by comparing real-time delivery and non-real-time delivery. This chapter discussed the most important protocols which support this service, the RTP/RTCP protocols. Since the RTP protocol is responsible for sending and receiving RTP packets and the RTCP protocol is responsible for monitoring network quality by its control packets, the benchmark suits consists of three elements, the RTP Sender benchmark, the RTP Receiver benchmark and the RTCP Processing benchmark. Each benchmark was discussed from three aspects on its implementation: function, measurement and environment.

4 Benchmarks Results

Previous *chapters reviewed major protocols used for the Internet by concentrating on the RTP/RTCP protocols used for real-time delivery, and presented the implementations of the RTP and RTCP benchmarks. This chapter describes the results of the architectural performance analysis by focusing on the following three benchmarks investigated in RTP and RTCP processing: the RTP Sender benchmark, the RTP Receiver benchmark, and the RTCP Processing benchmark. Before simulating these benchmarks, several assumptions have to be made. Subsequently, these benchmarks are run on the cycle-accurate* sim-outorder *simulator from the SimpleScalar tool set (Version 3.0) in order to obtain the results for each benchmark. Specifically, the results are presented from two aspects. First, profiling results presents the number of clock cycles for executing each function in each benchmark in comparison to the total cycles for executing the benchmark. These results highlight the performance of the benchmarks and allow to easily find the time-critical functions in the benchmarks. Second, the results on architectural characteristics of the benchmarks are determined and compared with the results from NetBench and MediaBench by looking at instruction level parallelism, branch prediction accuracy, instruction distribution and cache behavior.*

This chapter is organized as follows. Section 4.1 presents the assumptions we made in order to simulate the benchmarks. Section 4.2 presents the RTP Sender benchmark results. Section 4.3 presents the RTP Receiver benchmark results. Section 4.4 presents the RTCP Processing benchmark results. Section 4.5 presents the conclusions of this chapter.

4.1 Assumptions

In this section, we discuss the assumptions made prior to running any simulations in our architectural analysis.

- We utilized the following data set for the benchmarks:
  - For the RTP Sender benchmark, we utilized an audio file as input data. The size of the file is *4MB*. This input was chosen to simulate the data what would have been generated when talking into a microphone. Three different outputs of stream of packets are generated by this benchmark. The first output data consists of an in-order packet stream where the sequence numbers of packets are in the order (called $S\_output1$). The second output data consists of an out-of-order packet stream where the sequence numbers of packets are slightly out-of-order (called $S\_output2$). The third output data reflects the worst network

environment where the sequence numbers of packets are largely out-of-order (called *S_output*3).

- For the RTP Receiver benchmark, we utilized the output of stream of packets produced by the RTP Sender benchmark as input data. The input data consists of an in-order packet stream where the sequence numbers of packets are in the order (called *R_input*1). In order to better simulate a more realistic networking environment, in which packets may arrive out of order. Two additional packet streams were generated based on the previously mentioned in-order one. Therefore, the second input consists of an out-of-order packet stream where the sequence numbers of packets are slightly out of order (called *R_input*2). The third input data reflect an even more erratic network environment where the sequence numbers of packets are largely out-of-order (called *R_input*3).
- For the RTCP Processing benchmark, we have taken the same data set as the RTP Receiver benchmark.

- We assume the benchmarks work for the voice data delivery in a unicast environment.

- We have utilized a cycle accurate simulator called *sim-outorder* from the SimpleScalar tool set (V3). The simulator entails a 2-way superscalar processor with *64 KB* of direct-mapped level *1* (*L1*) data and instruction caches and a *1 MB* unified level *2* (*L2*) caches. The *L1* and *L2* cache latencies are set to *1* and *6* cycles, respectively.

- The base machine comprises the following units: 4 integer ALUs, 1 integer MULT/DIV unit, 4 floating-point adders, 1 floating-point MULT/DIV unit, and 2 memory ports (Read/Write).

- We compiled the benchmarks utilizing modified GNU binary utilities that specifically target the MIPS architecture used in the SimpleScalar tool set.

- We utilized the metrics defined in Chapter 3 to evaluate the benchmarks. These metrics highlight two aspects, namely performance and architectural characteristics:

  - Performance is measured by counting the clock cycles spent in each function in comparison to the total number of clock cycles it takes to execute the benchmarks.
  - The architectural characteristics are determined by looking at instruction level parallelism, branch prediction accuracy, instruction distribution and cache behavior.

These assumptions are made for the benchmarks in order to achieve the simulation results. In the following sections, the results for each benchmark are discussed, respectively.

4.2 The RTP Sender Benchmark Results

In this section, we discuss the RTP Sender benchmark results. We first present the performance results in clock cycles. The profiling is performed by determining the percentage of clock cycles spent by each function in the benchmark compared to the total clock cycles. Subsequently, the results on architectural characteristics are presented and compared with the applications from NetBench and MediaBench [9].

Profiling results: The performance results in clock cycles for different output files are depicted in Figure 4.1. Three output files are written to the hard disk, in-order writing ($S\_output1$ depicted in 4.1(a)), slightly out-of-order writing ($S\_output2$ depicted in 4.1(b)) and largely out-of-order ($S\_output3$ depicted in 4.1(c)). The percentage of the number of cycles for the encapsulation function (encap), which is the main function of the benchmark, is decreased for the three issues depicted in 4.1(a), 4.1(b) and 4.1(c), respectively, because the number of total cycles for these three issues are increased, which are caused by increased the number of cycles for the output function. As a matter of the fact, the number of cycles for the encapsulation function has not changed. Because the output function is the post processing of the benchmark, the result of which is used for the RTP Receiver benchmark and the RTCP Processing benchmark, it is reasonable that we ignore its cycles in the RTP Sender benchmark. Figure 4.1(d) depicts the performance result of the RTP Sender benchmark without the output function. From this figure, we can observe that the number of cycles for the encapsulation function is more than 70% of the total cycles.

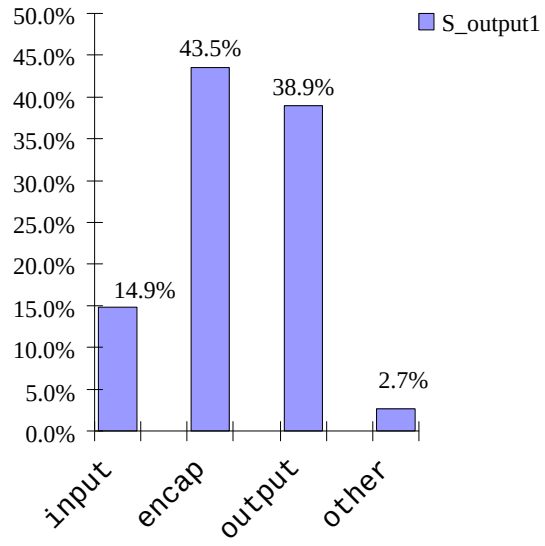
Results on architectural characteristics: Table 4.1[1] presents the simulation results on architectural characteristics: Table 4.1(a) presents the results on instruction level parallelism and branch prediction accuracy, Table 4.1(b) presents the results on instruction distribution, and Table 4.1(c) presents the results on cache behavior. The results are generated by simulating the benchmark where the output function is not included. Comparing with the application from NetBench and MediaBench depicted in the last two rows of Table 4.1[2], we can observe:

- Table 4.1(a) shows that the IPC of the RTP Sender is 19.8% and 37.2% higher than NetBench and MediaBench, respectively. The APR of the RTP Sender is 1.17% and 6.6% higher than NetBench and MediaBench, respectively. The DPR of the RTP Sender is 0.96% and 5.8% higher than NetBench and MediaBench, respectively.

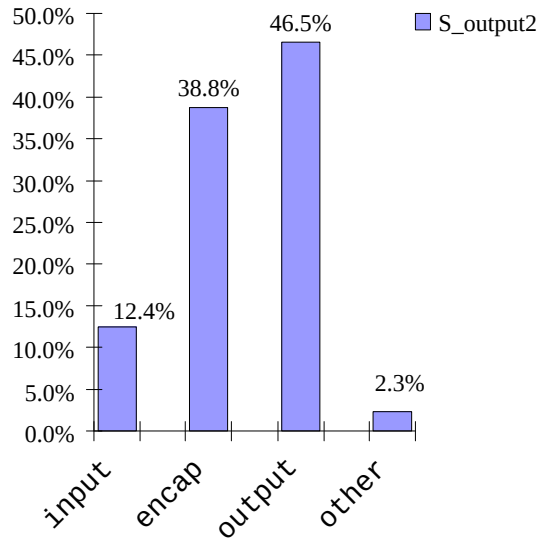
- Table 4.1(b) shows that the RTP Sender has a higher load/store instruction frequency than NetBench and MediaBench.

[1] The abbreviations in Table 4.1 are described in Section 3.2.4 of Chapter 3.

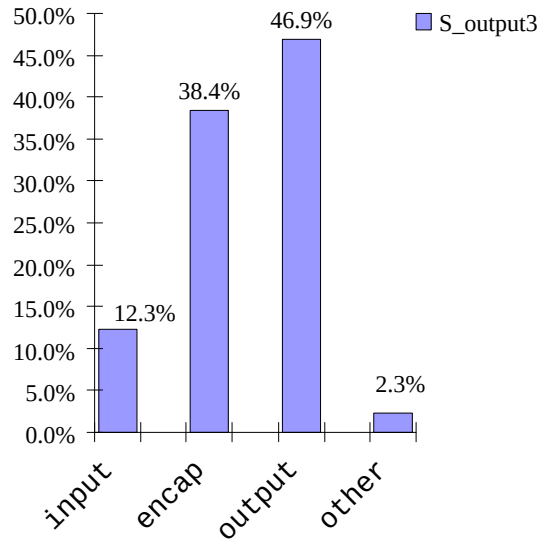
[2] The results on NetBench and MediaBench are the averages of the results presented in [9] due to the large mix of different small and big benchmarks. We are able to use the averages over the results, because there are no significantly big differences between the results, except for the number of cycles (*# of cycles*), the number of instructions (*# of inst.*), and the number of il1 and dl1 accesses (*il1 acc.* and *dl1 acc.*). Therefore, no comparison is made based on these values and they were only shown for completeness.



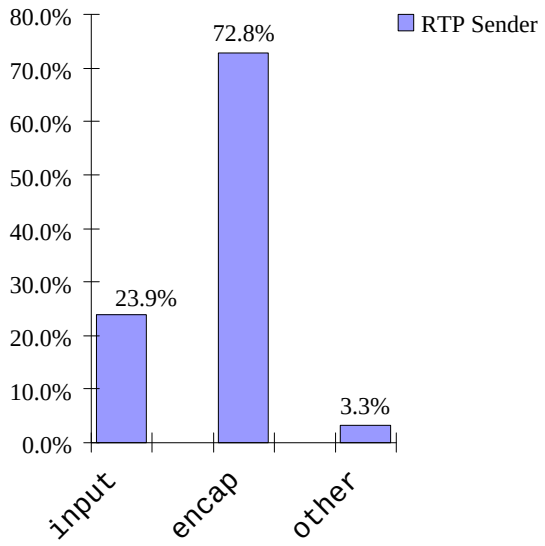
(a) The RTP Sender benchmark result for the in-order writing issue.



(b) The RTP Sender benchmark result for the slightly out-of-order writing issue.



(c) The RTP Sender benchmark result for the largely out-of-order writing issue.



(d) The RTP Sender benchmark result without the output function.

Figure 4.1: The RTP Sender benchmark results

	# of cycles (M)	IPC	APR (%)	DPR (%)
RTP Sender	11.4	1.99	95	95.1
NetBench	207	1.66	93.9	94.2
MediaBench	280	1.45	89.1	89.9

(a) IPC and branch prediction values for the RTP Sender benchmark.

	# of inst. (M)	load (%)	store (%)	branch (%)
RTP Sender	22.8	31.3	21.9	12
NetBench	359	27.7		7.2
MediaBench	408	19.8		11.3

(b) Instruction distribution for the RTP Sender benchmark.

	il1 acc. (M)	il1 miss ratio (%)	dl1 acc. (M)	dl1 miss ratio (%)	l2 miss ratio (%)
RTP Sender	23.9	0.0	11.9	0.0	4.6
NetBench	400	0.05	140	0.8	9.7
MediaBench	519	0.4	86	1.8	14.8

(c) Cache behavior for the RTP Sender benchmark.

Table 4.1: The architectural characteristics for the RTP Sender benchmark.

- Table 4.1(c) shows that the RTP Sender has a better performance in cache behavior than NetBench and MediaBench because the il1 miss ratio, dl1 miss ratio and l2 miss ratio of the RTP Sender are less than the ones of NetBench and MediaBench.

4.3 The RTP Receiver Benchmark Results

In this section, we discuss the RTP Receiver benchmark results. We first present the performance results in clock cycles. The profiling is performed by determining the percentage of clock cycles spent by each function in the benchmark compared to the total clock cycles. Subsequently, the results on architectural characteristics are presented and compared with the applications from NetBench and MediaBench.

Profiling results: The performance results in the clock cycles for different input files are depicted in Figure 4.2. Three input files are used from the hard disk, in-order ($R\_output1$ depicted in 4.2(a)), slightly out-of-order ($R\_output2$ depicted in 4.2(b)) and largely out-of-order ($R\_output3$ depicted in 4.2(c)), which are the output of the RTP Sender benchmark. Figure 4.2(d)[3] is the collection of Figure 4.2(a), 4.2(b) and 4.2(c). From the results presented in Figure 4.2, we can observe:

- The biggest contributor to the total cycles is the input function, which is caused by reading the data in the hard disk.
- The percentage of the number of cycles for the queueing function is increased from about 11% to 18% for the three issues presented in Figure 4.2(a), 4.2(b) and 4.2(c), respectively. The increase is reasonable because the in-order processing is obviously less than the out-of-order processing and the slightly out-or-order processing is

[3]Only the biggest value in each function is marked in Figure 4.2(d).

obviously less than the largely out-of-order processing[4]. However, the increase range, 7%, is not the maximum range. As a matter of fact, we can not obtain such maximum value. This observation signifies that the queueing operation could largely contribute to the overall processing when the packets come in a largely out-of-order fashion.

- The percentage of the number of cycles for the three functions, buffer, parsing, and updating, is slightly decreased between 1% and 2% for the three issues presented in Figure 4.2(a), 4.2(b) and 4.2(c), respectively. As a matter of fact, the absolute number of cycles for these three functions in comparison to the total clock cycles has not changed, while the number of total cycles are increased because the number of cycles for the queueing function is increased.

Figure 4.3 presents the performance results for three different issues without the input function in the benchmark. The input function can be ignored, because it is the preprocessing of the benchmark. Figure 4.3(d) is the collection of Figure 4.3(a), Figure 4.3(b) and Figure 4.3(c). From the results of Figure 4.3, we can observe that the biggest contributor to the the total cycles is the statistics updating function, averaging about 25%. The second largest contributor is the data parsing function, averaging about 23%.

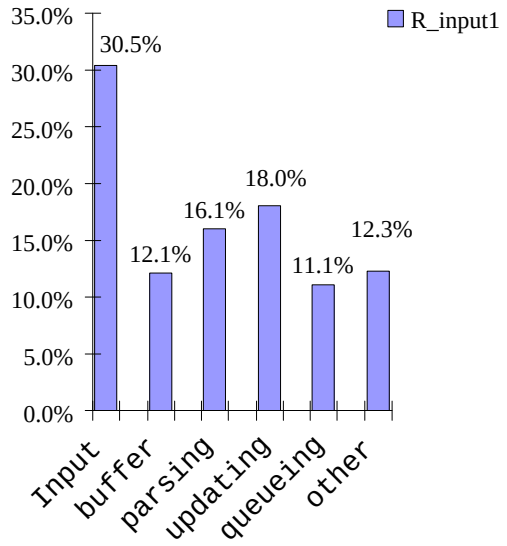
Results on architectural characteristics: Table 4.2 presents the simulation results on architectural characteristics: Table 4.2(a) presents the results on instruction level parallelism and branch prediction accuracy, Table 4.2(b) presents the results on instruction distribution, and Table 4.2(c) presents the results on cache behavior. In Table 4.2, R_input1 denotes the in-order input file used in the simulation, R_input2 denotes the slightly out-of-order input file used in the simulation, and $R\_output3$ denotes the largely out-of-order input file used in the simulation. Average is the arithmetic mean of the three issues. Comparing with the NetBench and MediaBench applications depicted in the last two rows of Table 4.2, we can observe:

- Table 4.2(a) shows that the IPC of the RTP Receiver is 14.5% and 31% higher than NetBench and MediaBench, respectively. The APR of the RTP Receiver is 3.4% higher than MediaBench. The DPR of the RTP Receiver is 2.6% higher than MediaBench.

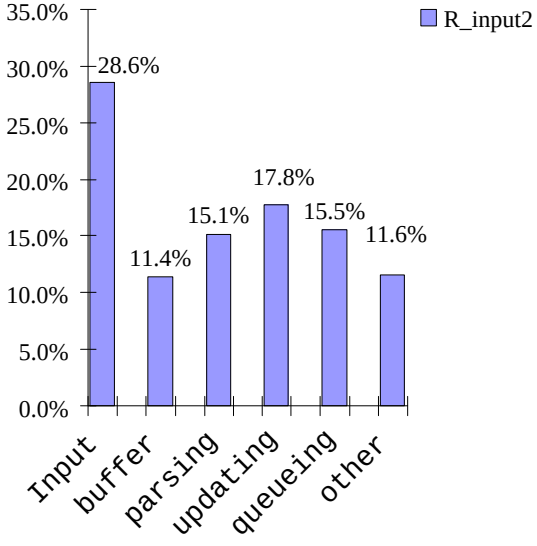
- Table 4.2(b) shows that the RTP Receiver has a higher load/store instruction frequency than both NetBench and MediaBench.

- Table 4.2(c) shows that the RTP Receiver has a better performance in cache behavior than NetBench and MediaBench because the il1 miss ratio, dl1 miss ratio and l2 miss ratio of the RTP Receiver are less than the ones of NetBench and MediaBench.

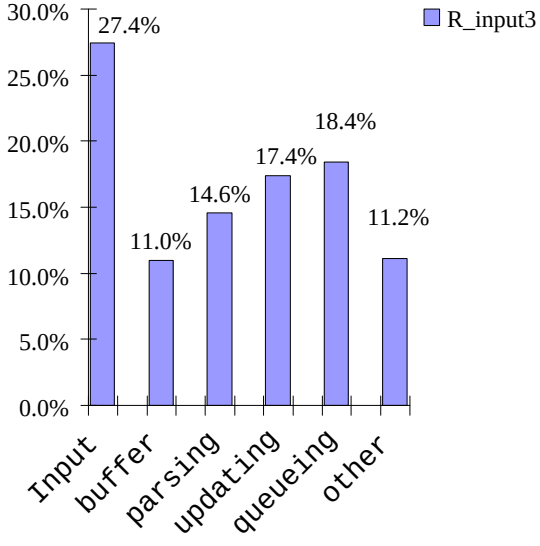
[4]The algorithm for the queue processing is presented in Chapter 3.



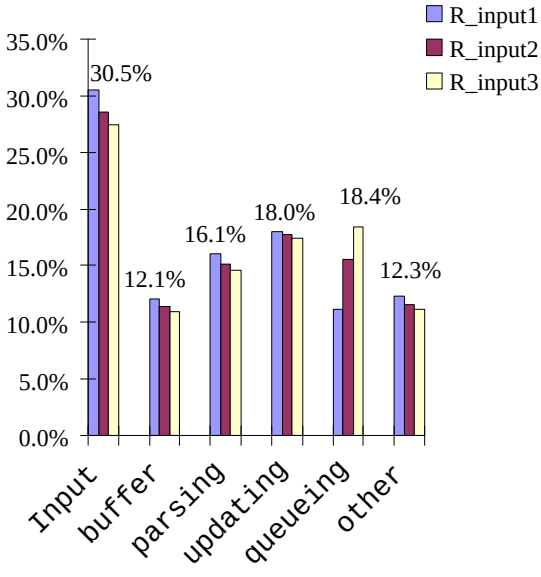
(a) The RTP Receiver benchmark result for the in-order issue.



(b) The RTP Receiver benchmark result for the slightly out-of-order issue.

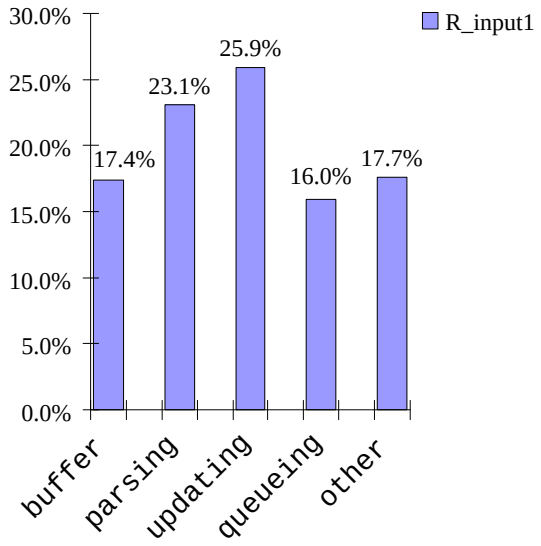


(c) The RTP Receiver benchmark result for the largely out-of-order issue.

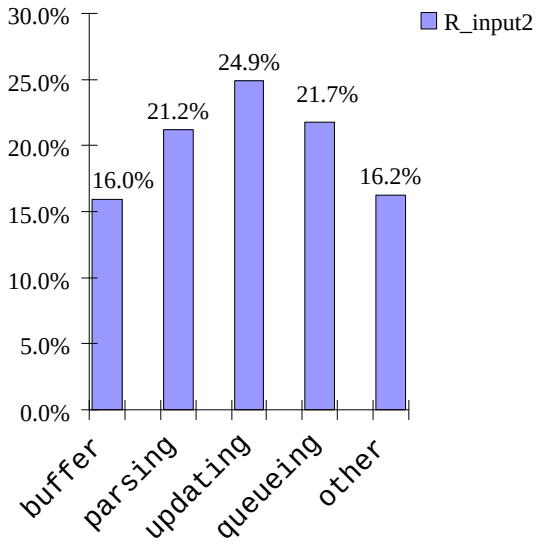


(d) The combined RTP Receiver benchmark results.

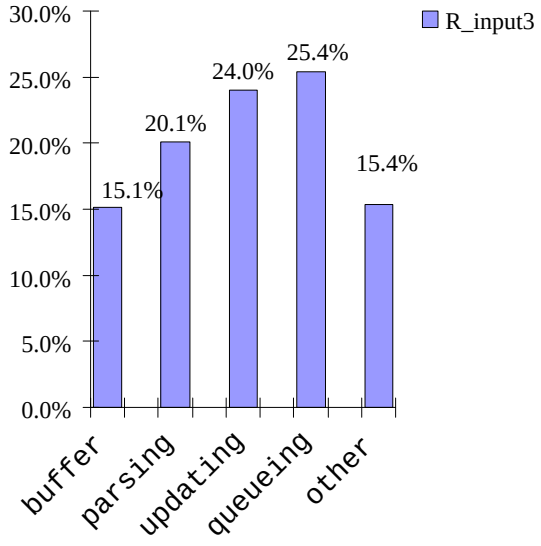
Figure 4.2: The RTP Receiver benchmark results with the input function.



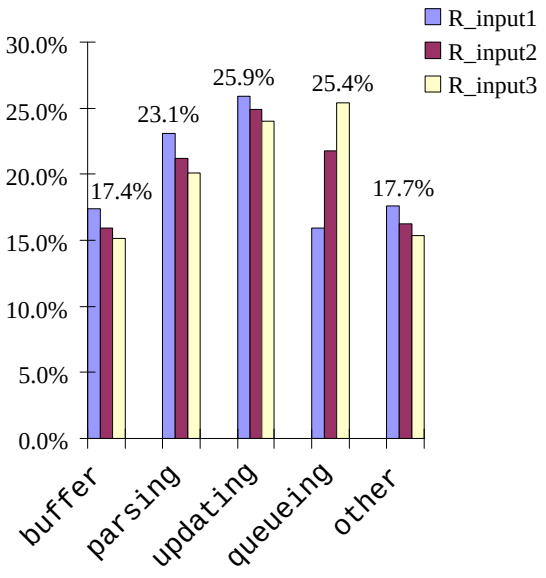
(a) The RTP Receiver benchmark results for the in-order issue.



(b) The RTP Receiver benchmark result for the slightly out-of-order issue.



(c) The RTP Receiver benchmark result for the largely out-of-order issue.



(d) The combined RTP Receiver benchmark results.

Figure 4.3: The RTP Receiver benchmark results without the input function.

	# of cycles (M)	IPC	APR (%)	DPR (%)
R. input1	15.7	1.97	92.5	92.5
R. input2	16.6	1.92	91.9	91.9
R. input3	17.2	1.92	92.3	92.3
Average	16.5	1.9	92.2	92.2
NetBench	207	1.66	93.9	94.2
MediaBench	280	1.45	89.1	89.9

(a) IPC and branch prediction values for the RTP Receiver benchmark.

	# of inst. (M)	load (%)	store (%)	branch (%)
R. input1	30.8	29.1	18.9	14.1
R. input2	31.9	30	18.8	14.3
R. input3	33	30.9	18.7	14.5
Average	31.9	30.0	18.8	14.3
NetBench	359	27.7		7.2
MediaBench	408	19.8		11.3

(b) Instruction distribution for the RTP Receiver benchmark.

	il1 acc. (M)	il1 miss ratio (%)	dl1 acc. (M)	dl1 miss ratio (%)	l2 miss ratio (%)
R. input1	32.4	0.0	14.3	0.4	0.8
R. input2	34.1	0.0	14.9	0.4	0.8
R. input3	35.3	0.0	15.6	0.4	0.8
Average	33.9	0.0	14.9	0.4	0.8
NetBench	400	0.05	140	0.8	9.7
MediaBench	519	0.4	86	1.8	14.8

(c) Cache behavior for the RTP Receiver benchmark.

Table 4.2: The architectural characteristics for the RTP Receiver benchmark.

4.4 The RTCP Processing Benchmark Results

In this section, we discuss the RTCP Processing benchmark results. We first present the performance results in clock cycles. The profiling is performed by determining the percentage of clock cycles spent by each function in the benchmark compared to the total clock cycles. Subsequently, the results on architectural characteristics are presented and compared with the applications from NetBench and MediaBench.

Profiling results: The performance results in clock cycles are depicted in Figure 4.4. We can observe that the biggest contributor to the total cycles is building RTCP receiver report (block), which percentage to the total cycles[5] is about 55%.

Results on architectural characteristics: Table 4.3 presents the simulation results on architectural characteristics: Table 4.3(a) presents the results on instruction level parallelism and branch prediction accuracy, Table 4.3(b) presents the results on instruction distribution, and Table 4.3(c) presents the results on cache behavior. Comparing with the NetBench and MediaBench applications depicted in the last two rows of Table 4.3, we can observe:

- Table 4.3(a) shows that the IPC of the RTCP Procesessing is 10.8% and 26.9% higher than NetBench and MediaBench, respectively. The APR and the DPR of the RTCP Processing is 4.7% and 3.8% higher than MediaBench, respectively.

[5]The value of the total clock cycles is not equal to 33M, depicted in Table 4.3(a), which includes the number of cycles of the RTP Receiver application because the RTCP Processing has to work in conjunction with the RTP Receiver benchmark.

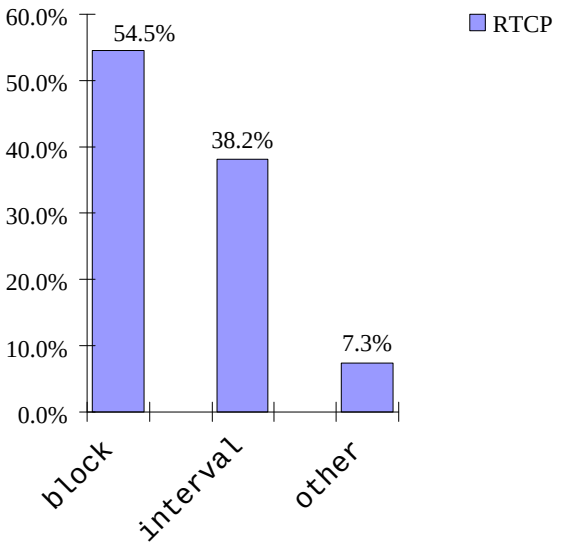


Figure 4.4: The RTCP Processing benchmark results.

- Table 4.3(b) shows that the RTCP Processing has a higher load/store instruction frequency than NetBench and MediaBench. The RTCP Processing also has a lower branch instruction frequency than NetBench and MediaBench.
- Table 4.3(c) shows that the RTCP Processing has a better performance in cache behavior than NetBench and MediaBench because the il1 miss ratio, dl1 miss ratio and l2 miss ratio of the RTCP Processing are less than the ones of NetBench and MediaBench.

4.5 Conclusions

The results for the RTP/RTCP benchmarks were evaluated on two aspects: performance and architectural characteristics. First, the performance results presented the number of clock cycles for each function in each benchmark in relation to the total number of clock cycles. This evaluation allowed us to perform profiling in order to determined the time-critical functions in the benchmarks. Second, the results on the architectural characteristics presented by looking at instruction level parallelism, branch prediction accuracy, instruction distribution, and cache behavior. These results were compared with the applications from NetBench and MediaBench. The four characteristics are highlighted in Table 4.4: Table 4.4(a) highlights the results on instruction level parallelism and branch prediction accuracy, Table 4.4(b) highlights the results on instruction distribution, and Table 4.4(c) highlights the results on cache behavior. In Table 4.4, RTP Receiver is the average value taken from Table 4.2 and Average is the arithmetic mean

	# of cycles (M)	IPC	APR (%)	DPR (%)
RTCP	23.6	1.84	93.3	93.3
NetBench	207	1.66	93.9	94.2
MediaBench	280	1.45	89.1	89.9

(a) IPC and branch prediction values for the RTCP Processing benchmark.

	# of inst. (M)	load (%)	store (%)	branch (%)
RTCP	43.6	29	17.2	1.3
NetBench	359	27.7		7.2
MediaBench	408	19.8		11.3

(b) Instruction distribution for the RTCP Processing benchmark.

	il1 acc. (M)	il1 miss ratio (%)	dl1 acc. (M)	dl1 miss ratio (%)	l2 miss ratio (%)
RTCP	45.1	0.0	19.2	0.3	0.8
NetBench	400	0.05	140	0.8	9.7
MediaBench	519	0.4	86	1.8	14.8

(c) Cache behavior for the RTCP Processing benchmark.

Table 4.3: The architectural characteristics for the RTCP Processing benchmark.

	# of cycles (M)	IPC	APR (%)	DPR (%)
RTP Sender	11.4	1.99	95	95.1
RTP Receiver	16.5	1.9	92.2	92.2
RTCP	33	1.84	93.3	93.3
Average	20.3	1.9	93.5	93.5
NetBench	207	1.66	93.9	94.2
MediaBench	280	1.45	89.1	89.9

(a) Comparison in IPC and branch prediction values between RTP/RTCP benchmarks with NetBench and MediaBench.

	# of inst. (M)	load (%)	store (%)	branch (%)
RTP Sender	22.8	31.3	21.9	12
RTP Receiver	31.9	30.0	18.8	14.3
RTCP	43.6	29	17.2	1.3
Average	32.8	30.1	19.3	9.2
NetBench	359	27.7		7.2
MediaBench	408	19.8		11.3

(b) Comparison in instruction distribution between RTP/RTCP benchmarks with NetBench and MediaBench.

	il1 acc. (M)	il1 miss ratio (%)	dl1 acc. (M)	dl1 miss ratio (%)	l2 miss ratio (%)
RTP Sender	23.9	0.0	11.9	0.0	4.6
RTP Receiver	33.9	0.0	14.9	0.4	0.8
RTCP	45.1	0.0	19.2	0.3	0.8
Average	34.3	0.0	15.3	0.2	2.1
NetBench	400	0.05	140	0.8	9.7
MediaBench	519	0.4	86	1.8	14.8

(c) Comparison in cache behavior between RTP/RTCP benchmarks with NetBench and MediaBench.

Table 4.4: Comparison in the architectural characteristics between RTP/RTCP benchmarks with NetBench and MediaBench.

of the RTP Sender benchmark, the RTP Receiver benchmark and the RTCP Processing benchmark. The performance analysis results show that:

- The biggest contributor to the total cycles for the RTP Sender benchmark is the encapsulation function, which is more than 70% of the total cycles. The first two biggest contributors for the RTP Receiver benchmark are the statistics updating

function and the data parsing function, which are about 25% and 23% of the total cycles, respectively. The biggest contributor for the RTCP Processing benchmark is building RTCP receiver block, which takes about 55% of the total RTCP Processing cycles.

- The results depicted in Table 4.4(a) show that the average IPC of the RTP/RTCP benchmarks is 14.5% and 31% higher than NetBench and MediaBench, respectively. The average APR of the RTP/RTCP benchmarks is 4.9% higher than MediaBench. The average DPR of the RTP/RTCP benchmarks is 4% higher than MediaBench. The results depicted in Table 4.4(b) show that the average load/store instruction frequency of the RTP/RTCP benchmarks is higher than NetBench and MediaBench. The average branch instruction frequency of the RTP/RTCP benchmarks is lower than MediaBench. The results depicted in Table 4.4(c) show that the RTP/RTCP benchmarks has a better performance in the first level data/instruction caches and in the second level unified data/instruction cache than NetBench and MediaBench.

In conclusion, the results on architectural characteristics show us that the RTP/RTCP processing is significantly different from the media processing (MediaBench), which means that it is necessary to create benchmarks for network processing. The results also show us that the RTP/RTCP processing has some common characteristics with NetBench. It is reasonable since they all focus on the same network processing domain. However, they are slight different on some characteristics with each other, which means that the specific benchmarking suite in the RTP/RTCP processing is justified.

5 Conclusions

It has been *argued in this thesis that the latest developments of the Internet create two requirements on network devices: performance and flexibility. In order to meet the two requirements, a new kind of processor has emerged namely the network processor (NP). It is specifically designed to process data at wire-speed and to be flexible to support new applications. However, the existence of different architectures in network processors makes it more difficult to evaluate the performance of the network processors. Consequently, it is necessary to create benchmarks that allow the performance of network processors to be evaluated. This thesis has highlighted that benchmarking on real-time network processing is important since as an example, Voice over IP (VoIP) is becoming an increasingly promising technology. In order to provide real-time delivery services, the RTP/RTCP protocols are utilized. To this end, the benchmark suite on the RTP/RTCP processing has been created and described in this thesis. The benchmark suite consists of three benchmarks: an RTP Sender benchmark, an RTP Receiver benchmark and an RTCP Processing benchmark. For each benchmark, three aspects were specified: function, measure and environment. The function aspect specified the core algorithms used in each benchmark, all necessary functions realized in the benchmark and the manner to implement these functions. The measurement aspect specified the verification of the benchmarks and the metrics used to investigate the benchmarks. The environment aspect specified the input/output of each benchmark and simulation environment of the benchmarks. Finally, the results for the benchmarks were evaluated from performance and architectural characteristics points of view. First, the performance results presented the clock cycles spent on each function in each benchmark in relation to the total cycles. Subsequently, profiling on each benchmark was performed in order to find the time-critical functions in the benchmarks. Second, the results on architectural characteristics were investigated and compared with the applications from NetBench and MediaBench in order to understand the features of the benchmarks. The investigated characteristics are instruction level parallelism, branch prediction accuracy, instruction distribution, and cache behavior.*

This chapter provides some concluding remarks, highlights the main contributions of this thesis, and presents some possible future research directions. This chapter is organized as follows. Section 5.1 summarizes the main conclusions of this thesis. Section 5.2 presents the main contributions of this thesis. Section 5.3 highlight several possible future research directions.

5.1 Summary

Chapter 2 discussed the TCP/IP model, explained the main protocols in the TCP/IP model and analyzed the underlying functions in network processing. Network processing consists of a wide range of functions, such as IP routing, encryption, and real-time delivering. We argued in this chapter that either general-purpose processor or ASICs alone can hardly possible meet the two requirements: performance and flexibility. General-purpose processors are more flexible to newer protocols with lower performance while ASICs have a higher performance for network processing with lower flexibility. New solutions are called the network processors, which take into account the two requirements and the trade-off between them. Currently, there are a number of network processors with different architectures. In order to evaluate the performance of the different network processors, benchmarks are needed. We discussed some existing network processing benchmarks and highlighted that these benchmarks do not cover all applications in the networking domain. Subsequently, it was argued that it is necessary and important to create benchmarks on some untargeted applications. Finally, we briefly presented an overview on the SimpleScalar tool set utilized for simulating the created benchmarks in this thesis.

Chapter 3 compared differences between real-time delivery and non-real-time delivery and discussed the protocols for supporting real-time delivery: the RTP/RTCP protocols. We highlighted that the current IP network only provides "best-effort" services in non-real-time delivery. Packets to the destination may be delayed or out of order, or packets may be lost before reaching the destination. Real-time delivery, however, is intolerant to delay and jitter. In order to solve the two problems: jitter and order, the RTP/RTCP protocols are utilized to ensure real-time delivery.

In this chapter, we discussed the implementation of three benchmarks on RTP/RTCP processing, an RTP Sender benchmark, an RTP Receiver benchmark, and an RTCP Processing benchmark. First, the RTP Sender benchmark investigates how RTP packets are generated at the sender. We discussed the main functions in the RTP Sender benchmark and defined the input and output of this benchmark. Specifically, we implemented a special structure utilized for writing the output of this benchmark into the hard disk in order for the output of the RTP Sender benchmark to be utilized as the inputs of the other two benchmarks. Second, the RTP Receiver benchmark investigates how RTP packets are processed at the receive in order to handle the order and jitter problems. In this benchmark, we designed four functions for processing every coming RTP packet: input buffer management, data parsing, statistics updating and queue management. Additionally, we defined the input and output of this benchmark. Third, the RTCP Processing benchmark generates receiver reports for feedback and computes the RTCP transmission interval for the receiver to adapt its rate of reporting. Similarly, we also defined the input and output of the RTCP Processing benchmark.

The three created benchmarks were presented from three aspects: function, measurement and environment. First, the function aspects specified the main functions and

their implementations in the benchmarks. Second, the measurement aspect specified the metrics utilized for evaluating the benchmarks. In order to measure the performance of the benchmarks, we defined the clock cycles as the performance metric. In addition, we also defined the architectural characteristics as our metrics in order to understand the features of the benchmarks. These characteristics include instruction level parallelism, branch prediction accuracy, instruction distribution, and cache behavior. Third, the environment aspects specified the interface (input/output) environment of each benchmark and the simulation environment of the benchmarks. We highlighted that the same measurement metrics and simulation environment are utilized in the three benchmarks.

Chapter 4 presented the results after running the RTP/RTCP benchmarks utilizing the *sim-outorder* simulator. The results were evaluated from performance and architectural characteristics points of view. First, the performance results presented the number of clock cycles for each function in each benchmark in relation to the total number of clock cycles. The profiling was performed and presented for each benchmark in order to find the time-critical functions in the benchmarks. Second, the results on the architectural characteristics presented by looking at instruction level parallelism, branch prediction accuracy, instruction distribution, and cache behavior. These results were compared with the applications from NetBench and MediaBench. A brief overview on the results is presented as follows:

- **Profiling results:** The biggest contributor to the total cycles for the RTP Sender benchmark is the encapsulation function, which is more than 70% of the total cycles. The first two biggest contributors for the RTP Receiver benchmark are the statistics updating function and the data parsing function, which are about 25% and 23% of the total cycles, respectively. The biggest contributor for the RTCP Processing benchmark is building RTCP receiver report block, which takes about 55% of the total RTCP processing cycles.

- **Results on architectural characteristics:** On instruction level parallelism, on average, the RTP/RTCP benchmarks have a higher data-level parallelism than NetBench and MediaBench, which is about 14.5% and 31%, respectively. On branch prediction accuracy, RTP/RTCP benchmarks on average allow a better address prediction accuracy (4.9% higher) and direction prediction accuracy (4% higher) than MediaBench. On instruction distribution, RTP/RTCP benchmarks have a higher load/store frequency than NetBench and MediaBench and a higher branch instruction frequency than MediaBench. On cache behavior, RTP/RTCP benchmarks have a better performance in the first level data/instruction caches and the second level unified data/instruction cache than NetBench and MediaBench.

In conclusion, the profiling results in Chapter 4 highlighted how fast, measured in terms of clock cycles, the benchmarks are executed and which functions in the benchmarks are time-critical. The results on architectural characteristics show that the RTP/RTCP processing is significantly different from the media processing (MediaBench), which means that it is necessary to create benchmarks on network processing. At the same time,

these results show that the RTP/RTCP processing is also different from other network processing (NetBench). Therefore, our benchmarks on real-time network processing are justified.

5.2 Main Contributions

In this section, we highlight the main contributions of this thesis to the design of network processors:

- We have created a benchmark suite on real-time network processing that has not been introduced before in existing benchmarks. This is driven by the trend that the convergence of voice with data will play an important roll in future networks. Therefore, we have investigated network processing on delivering voice data with real-time characteristics, and focused on the functionalities on the real-time network processing. This benchmark suite can be used to evaluate performance of real-time network processing.

- We have performed profiling on the benchmarks in order to determine which functions in the benchmarks are time-critical. The profiling results provide clear information that how fast, measured in terms of clock cycles, each functions in each benchmark are executed, what are the total clock cycles for the execution of the benchmarks and which functions in the benchmarks are time-critical. For the time-critical functions, we could achieve the performance gain by implementing on specialized hardware, which can be utilized in future network processors.

- We have investigated architectural characteristics on real-time network processing by focussing on instruction level parallelism, branch prediction accuracy, instruction distribution, and cache behavior. We have shown that these characteristics on real-time network processing are significantly different from media processing (MediaBench) and slightly different from other network processing (NetBench). The investigation can be utilized in the design of future network processor architectures.

In conclusion, the created benchmark suite in this thesis helps to measure and evaluate the performance of network processors and direct the design of future network processor architectures. The methodology utilized in this thesis can also be used to investigate benchmarks on some other network processing.

5.3 Future Research Directions

In this section, we present some future research directions in network processing benchmarking.

- This thesis introduces a benchmark suite on real-time network processing. Although RTP/RTCP is used for real-time delivery over IP networks, RTP/RTCP itself does not provide any mechanism to ensure timely delivery or provide other quality-of-service guarantees. To provide quality of service for real-time delivery,

signalling, such as SIP, should be used. Benchmarking SIP is a research topic of key interest.

- In this thesis, the benchmark suite is built in a unicast environment. However, RTP/RTCP can support real-time delivery to multiple destinations using multicast distribution. Benchmarking RTP/RTCP processing in the multicast environment and benchmarking multicast processing are promising topics to work on.

- In this thesis, the benchmark suite on real-time network processing is designed in software. The performance results indicated that some functions in the benchmarks take numerous clock cycles to execute. The performance gain can be achieved by implementing these functions in specialized hardware. Therefore, we believe that the time-critical functions should be implemented in actual hardware to measure the real performance gains in realistic scenarios.

Bibliography

[1] Tilman Wolf and Mark Franklin. CommBench - A Telecommunications Benchmark for Network Processors. `http://ccrc.wustl.edu/ wolf/cb/`, 1999.

[2] Burger, D. and Austin, Todd M. The Simplescalar Tool Set, Version 2. Technical report, University of Wisconsin, June 1997.

[3] Chunho Lee and Miodrag Potkonjak and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. `http://www.cs.ucla.edu/ leec/mediabench/`.

[4] Comer, Douglas E. *Internetworking with TCP/IP (Volumn 1)*. Prentice Hall, 2000.

[5] Comer, Douglas E. *Internetworking with TCP/IP (Volumn 3)*. Prentice Hall, 2000.

[6] Crowley, Patrick and Franklin, Mark A., editor. *Network Processor Design: issues and proctices (Volumn 1)* . Morgan Kaufman Publishers, 2003.

[7] J. Dunn. Methodology for ATM Benchmarking. In *RFC 3116*, June 2001.

[8] Embedded Microprocessor Benchmark Consortium. `http://eembc.org/`.

[9] Gokhan Memik, B. Mangione-Smith and W. Hu . NetBench: A Benchmarking Suite for Network Processors. Technical report, Compiler and Architecture Research Group (CARES) at University of California, November 2001.

[10] Jonathan Lennox, Jonathan Rosenberg and Dan Rubenstein, editor. *RTP Library*. Lucent Technologies, 1998.

[11] W.H. Mangione-Smith. Network Processors Technologies, December 2001.

[12] J. Perser. Benchmarking Methodology for LAN Switching Devices. In *RFC 2889*, 2000.

[13] Postel, J.B. Transmission Control Protocol. In *RFC 793*, August 1980.

[14] Postel, J.B. User Datagram Protocol. In *RFC 768*, August 1980.

[15] Prachant R. Chandra and Seow Yin Lim, editor. *Framework for Benchmarking Network Processor (Revision 1.0)*. Network Processing Forum, 2002.

[16] The Proceedings of the 2002 Workshop on Network Processors (NP-1). *Benchmarking Network Processors*, 2002.

[17] S. Bradner. Benchmarking Terminology for Network Interconnection Devices. In *RFC 1242*, 1991.

[18] S. Bradner. Benchmarking Methodology for Network Interconnect Devices. In *RFC 2544*, 1999.

[19] Schulzrinne, H. A Tranport Protocol for Real-Time Application. In *RFC 1889*, January 1996.

[20] Schulzrinne, H. RTP Profile for Audio and Video Conference with Minimal Control. In *RFC 1890*, January 1996.

[21] Niraj Shah. Understanding Network Processors, September 2001.

[22] SPEC. `http://www.specbench.org`.

[23] W.Richard Stevens. *TCP/IP Illustrated (Volumn 1).* Prentice Hall, 2000.

[24] Wayne Wolf. *Computers as Components: Principles of Embedded Computing System Design.* Morgan Kaufman Publishers, 2001.

Curriculum Vitae

Yunfei Wu was born in Suixi, China on the 12th of June 1974. In 1991, she was admitted into Nanjing University with exemption of national exams. She received her bachelor degree in Electronic Science Engineering at Nanjing University in July 1995. From 1995 to 2000, she worked as a teacher in Jiangsu Post and Telecommunication School (JPTS). After that, she worked as a switching design engineer in Dalian Ericsson Communication Company Nanjing Branch From 2000 to 2001.

In September 2001, she started her MSc study in Electrical Engineering at Delft University of Technology (TU Delft), The Netherlands. In September 2002, she started working on her MSc thesis at the Computer Engineering (CE) Laboratory under the supervision of Dr. Ir. Stephan Wong. The CE Laboratory is part of the Department of Electrical Engineering, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, and is chaired by Prof. Stamatis Vassiliadis. Her MSc thesis is entitled: "Benchmarking Real-Time Network Processing". Her research interests include: embedded systems design, advanced computer architectures, hardware/software co-design, digital signal processing, and telecommunication networks.