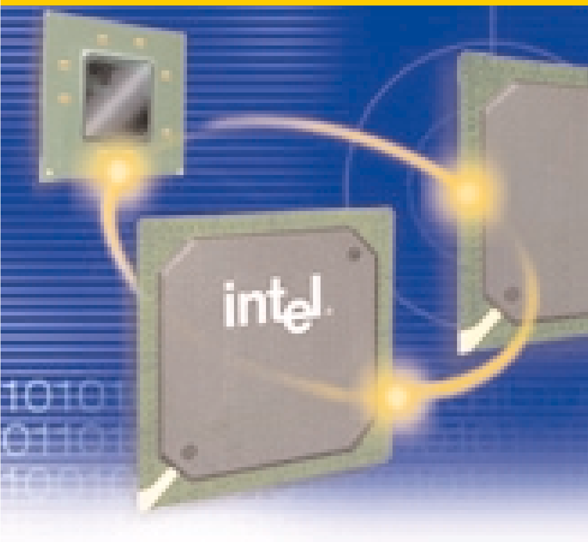
Volume 06 Issue 03 Published, August 15, 2002 ISSN 1535766X



Intel® Technology Journal

Network Processors

Implementing Voice over AAL2 on a Network Processor

A compiled version of all papers from this issue of the Intel Technology Journal can be found at:
http://developer.intel.com/technology/itj/index.htm

Implementing Voice over AAL2 on a Network Processor

Jaroslaw Sydir, Prashant Chandra, Alok Kumar,
Sridhar Lakshmanamurthy, Longsong Lin, Muthaiah Venkatachalam
Intel Communications Group, Intel Corporation

Index words: MEv2, IXP2400, IXA, network processor, VoAAL2, VoP, VoATM, VoIP, TM4.1, AAL2, AAL5, SAR

ABSTRACT

Programmable network processors are emerging as a versatile component for building telecommunications equipment because they can be programmed to perform a variety of different packet-processing functions, while allowing the equipment vendor to differentiate their products by supporting unique value-added features. A variety of applications with rather different characteristics and requirements can be deployed on a network processor. This ability to support a broad range of applications is one of the keys to the success of network processor products in the marketplace.

In this paper we describe a Voice over AAL2 (VoAAL2) processing application, as specified in International Telecommunications Union (ITU) recommendations I363.2 and I366.2. This application must satisfy the real-time requirements inherent in voice applications. It must use a jitter buffer and scheduler to remove jitter introduced in the network and a timer-based scheduler to ensure that voice packets do not incur too large of a processing delay. Also, because packets from low-data-rate voice channels are aggregated into high-data-rate Asynchronous Transfer Mode Virtual Circuits (ATM VCs) and vice versa, different components within the application must operate at different rates (some dealing with voice packets, others dealing with ATM cells). These requirements present some unique challenges to network processors that have traditionally been designed to support high-speed applications such as basic IP processing, where the processing of packets is very uniform and can be performed in a deterministically ordered pipeline. Applications such as VoAAL2, on the other hand, are a lot more asynchronous in nature. We discuss the requirements that applications like VoAAL2 place on network processor design and provide an example of how a VoAAL2 application can be implemented on the IXP2400 processor.

INTRODUCTION

Programmable Network Processors (NPUs) offer telecommunications equipment manufacturers a flexible platform for building a variety of different equipment. The power of NPUs is that they can be programmed to perform many different packet-processing functions to support a variety of different protocols and standards. This flexibility allows equipment manufacturers to utilize the same NPU or family of NPUs across different product lines. It also allows them to easily evolve their products to support evolving standards and to provide unique value-added features within these products.

Until recently, most network processors have been designed mainly to perform basic IP packet processing, as described in RFC 1812 [5], at very high line rates. This basic IP packet processing is fairly simple and deterministic. All packets are subject to roughly the same processing. Quality of Service (QoS) guarantees are either not provided at all or are provided on a course, per-class granularity with no guarantees on packet delay or packet delay variation (jitter).

Unfortunately, real-world packet-processing applications are much more complex and diverse than this basic IP processing application. IP processing performed by today’s routers includes many additional features such as DiffServ QoS, policy-based routing, and packet filtering that make the packet processing applications much more complex and less uniform. Other packet processing applications deal with real-time traffic and therefore have stricter real-time processing requirements.

In order to provide the level of flexibility sought by equipment manufacturers, an NPU must be able to support diverse packet-processing applications dealing with

connectionless and connection-oriented protocols with a variety of quality of service models and requirements. Different applications require different programming models, which must be supported by an NPU.

An example of an application that presents a different set of requirements than the basic IP forwarding application is the Voice over AAL2 (VoAAL2) application. In this paper we discuss the special requirements and challenges presented by the VoAAL2 application. We describe the architecture and design of a VoAAL2 application that we have developed for the Intel IXP2400 processor and discuss the features of an NPU that are required to support this type of application.

IXP2400 NETWORK PROCESSOR

The IXP2400 is a next-generation network processor developed by Intel Corporation. It is fully programmable, offering a very flexible programming model and support for a broad range of diverse packet processing applications. In this section we highlight some of the IXP2400 features that are utilized by the Voice over AAL2 (VoAAL2) application. “Network Processor Performance Analysis Methodology,” by Sridhar Lakshmanamurthy, et. al, provides a complete overview of the IXP2400 architecture [1].

The IXP2400 is a multi-threaded multi-processor system. Packets enter the IXP2400 through a configurable industry standard interface that supports Packet over SONET and UTOPIA interfaces. Packet processing is performed by eight packet-processing engines called MicroEngines (MEs). Each ME has eight hardware execution contexts (alternately referred to as threads). Each context has its own register set, so that swapping between them is a very fast (one instruction cycle) operation. The MEs use a non-preemptive context scheduling model, where the swapping out of contexts occurs under software control, and ready-to-run contexts are scheduled using a Round Robin scheduling discipline.

Each ME contain an Arithmetic Logic Unit (ALU) and a Content Associated Memory (CAM) unit, which allows the application to compare a key value to the keys of all CAM entries in one instruction cycle. The MEs also provide byte-alignment support to allow applications to manipulate packet headers and data that are not always aligned on four-byte boundaries. Each ME contains 640 longwords of local memory, where packet headers can be stored temporarily while they are processed. Finally, the MEs provide real-time timers, which allow a thread to specify a time in the future when it should be awakened.

The IXP2400 contains interfaces to external SRAM and SDRAM memories.

VOICE SERVICE REQUIREMENTS

Transmitting voice signals over a network places some specific real-time requirements on the network. Voice (for example, a telephone call) is sampled at periodic intervals, and those samples are transmitted across the network and replayed at the other end at the same rate. Each voice sample is subject to a transmission and propagation delay as it traverses the network. This delay cannot be too large if the conversation is to flow at a normal pace. More importantly, variations in the delay experienced by different samples (referred to as jitter) cannot cause the replay of those samples to occur at a variable rate, or the quality of the voice experience by the listener will degrade.

Traditional voice networks, Time Division Multiplexing (TDM) networks, solve this problem by reserving the capacity for the voice samples of a call at the time the call is set up and synchronizing the transmission of voice samples throughout the network. This provides a very predictable environment for voice applications and guarantees that the jitter and delay experienced by voice samples is within specified limits and yields the voice quality that we are used to when using the telephone. The drawback of this approach is that a 64Kb/s channel is reserved for the duration of the voice call and cannot be used to carry voice samples from other calls even during periods of silence.

Transmitting voice over packet networks, such as ATM, can solve this resource usage efficiency problem because voice samples from different calls are allowed to use the bandwidth from a call during periods of silence. The challenge is to allow this type of bandwidth sharing while still providing the same delay and jitter guarantees in order to maintain the same level of voice quality as a traditional voice network.

VOAAL2 SERVICE

Typical VoAAL2 Deployment

Figure 1 illustrates the typical configuration of a Voice over AAL2 (VoAAL2) service in a network. Voice calls originate on the Time Division Multiplexing (TDM) network.

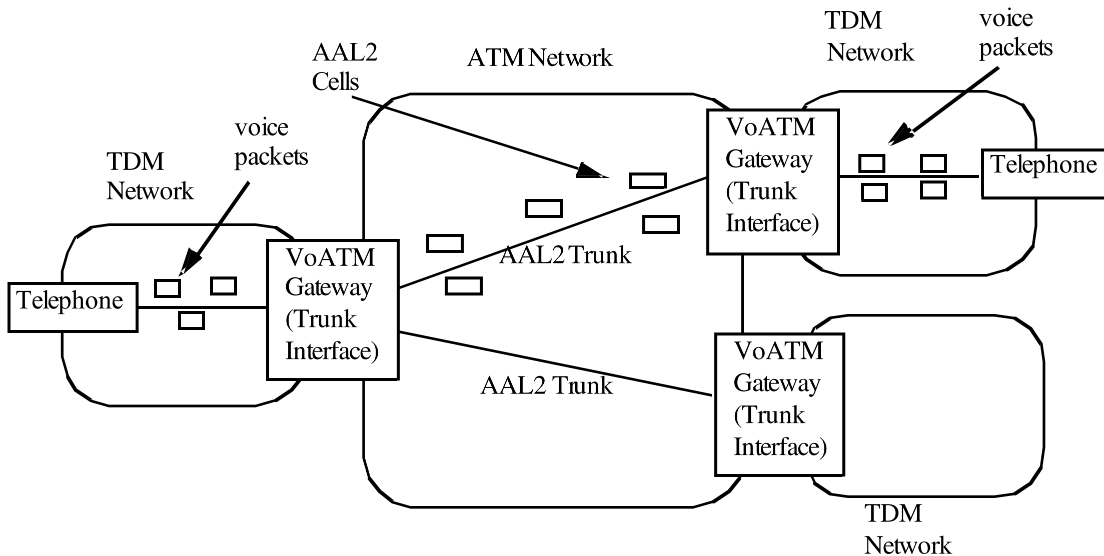


Figure 1: Configuration of a Voice over AAL2 (VoAAL2) service

The analog voice signal is sampled and digitized in the VoATM gateway to produce a stream of voice packets. (Note: TDM networks also carry digitized voice and other traffic.) These voice packets are transported across the ATM network in AAL2 trunks. An AAL2 trunk is an ATM Virtual Circuit (VC) used to transport AAL2 traffic. At the far end of the ATM network, the voice packets are converted back to an analog signal and transmitted over an analog voice network.

Figure 2 illustrates the relationship between voice calls, AAL2 channels, AAL2 Trunks, and ATM VCs. There are many Digital Signal Processors (DSPs) in the system. Each DSP processes a given number of voice calls. Each voice call is mapped to an AAL2 channel within an AAL2 trunk. There are 256 AAL2 channels within each AAL2 trunk. The AAL2 channel is identified by the Channel Identifier (CID). There are many AAL2 trunks in the system. Different voice calls from a given DSP can correspond to AAL2 channels within the same or different AAL2 trunks. The inverse relationship holds at the far end of the ATM network, where packets from each AAL2 channel are transformed into voice packets destined for a specific DSP. The VoAAL2 application transforms voice packets to AAL2 packets and transmits them on the correct AAL2 channel. At the other end of the ATM network, the VoAAL2 application performs the inverse operation.

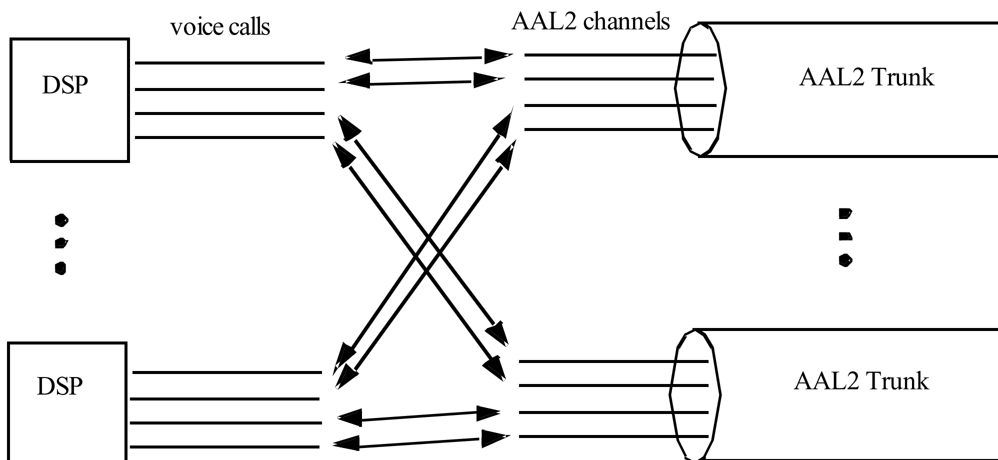


Figure 2: Relationship between voice calls and AAL2 channels

AAL2 Standards

The Voice over VoAAL2 service is specified by the International Telecommunications Union (ITU). AAL type 2 is subdivided into the Common Part Sublayer (CPS) and the Service-Specific Convergence Sublayer (SSCS). Recommendation I.363.2 specifies the CPS layer for all AAL type 2 applications [2]. This layer defines a packet format with a three-byte packet header, which contains the length of the packet, a User-to-User Indication (UUI) field, whose content is specified by the layer above, and a header error correction field. Recommendation I.366.2 specifies an SSCS layer for trunking of traffic from narrow band networks (ISDN or analog networks) over AAL2 [3]. I.366.2 defines a number of services for transporting audio and data traffic, and signaling over an AAL2 network. For the audio service, the SSCS layer does not define its own header. It simply specifies the format and values that are passed to the CPS layer and transmitted in the UUI field of the CPS header.

PACKET PROCESSING IN THE VOAAL2 APPLICATION

Figure 3 illustrates the relationship between the different types of packets. In the Digital Signal Processor (DSP) to Asynchronous Transfer Mode (ATM) direction, digitized voice packets are received from the DSP chip. Each voice packet is mapped to one Service-Specific Convergence Sublayer (SSCS)/Common Part Sublayer (CPS) packet. Multiple CPS packets are multiplexed within an AAL2 cell.

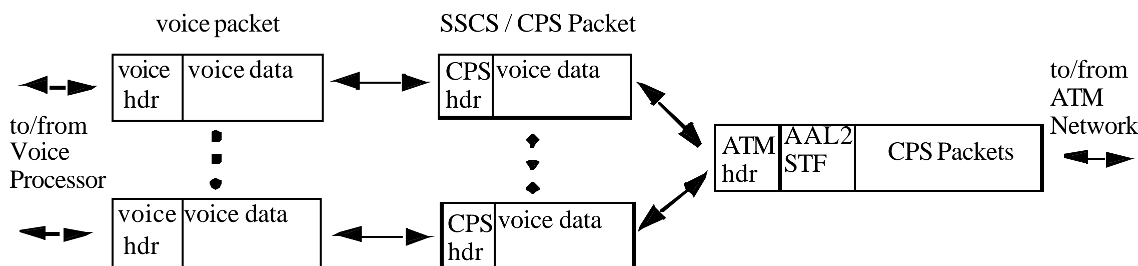


Figure 3: Voice over AAL2 packet transformations

The voice packets contain a header that indicates the identity of the voice channel, the length of the packet, and the encoding algorithm that was used. The voice header is a software convention established between the Voice over AAL2 (VoAAL2) application and the DSP and is not part of the standards. The information contained in this header could also be communicated by some out-of-band communication mechanism.

The application looks up the AAL2 channel, within a specific ATM Virtual Circuit (VC), that is used to transport this call. The DSP header is stripped off of the voice packet and SSCS processing is performed. SSCS processing involves generating the proper sequence number that is carried in the User-to-User Indication (UUI) field of the CPS header. This sequence number is maintained on a per-voice-call basis.

Next, CPS processing is performed. This entails the creation of the CPS header and the generation of a CRC-5-based header error correction field. Multiple CPS packets can be packed into the payload of an ATM cell, and the contents of a given CPS packet can be split across successive ATM cells (within the same VC). The first byte of AAL2 ATM cells contains a field called the Start Field (STF), which indicates the length of the data and offset to the start of the first CPS packet within the cell.

CPS packets destined for the same VC are accumulated until either a cell is completely filled or until the timer_CU has expired. Recommendation I.363.2 specifies the use of the timer_CU to force a cell to be sent after a certain amount of time, even if it is not full. The time_CU is used to make certain that the processing delay incurred by a CPS packet (voice packet) is less than a specified number. A timer_CU is maintained for each VC. When the first CPS packet is received (for a cell on that VC) the timer_CU is started for that VC. When additional CPS packets are received and the cell is filled, the timer_CU is canceled (or restarted if there is enough data to start a new cell). If the timer expires before the cell is full, the empty part of the cell is padded with zeros and the cell is sent even though it is not full.

The processing in the ATM to DSP direction is the inverse of the processing in the DSP to ATM direction. A cell is received on a particular VC. Within the cell are one or more CPS packets. The first packet may be a partial packet, part of which may have come in the previous cell on the same VC. Also, the last CPS packet may not be complete. The CPS packets within a cell can be destined for the same AAL2 channel or different channels.

The CPS packets are extracted from the ATM cell and reconstructed, and the CPS headers are verified using the CRC5 value in the header. For each packet the voice channel to which it belongs is determined as a function of the VC and AAL2 CID. The SSCS sequence number is also verified, and the packet is discarded if it is corrupted or misordered. The DSP header is prepended to the payload of the SSCS packet to create a DSP packet.

Each voice packet encodes a specified time interval of the voice signal. The timestamp for a packet represents the beginning of this interval. The SSCS sequence number captures the time of a packet relative to the time of the previous packet. The timestamp for packet n (Tn) is given by the formula: $Tn = Tn-1 + ((Sn - Sn-1)*I)$, where $Tn-1$ is the timestamp of packet $n-1$, Sn and $Sn-1$ are the sequence numbers of packets n and $n-1$ respectively, and I is the interval length for this call.

When a packet is received, its SSCS sequence number is used to generate a timestamp, which is used to perform jitter removal from the stream of voice packets that make up a call. Jitter is the variable inter-packet gap caused by network queuing and transmission delays experienced by successive packets or cells from one connection. Network jitter causes voice packets from a channel to be either bunched together or spread out in time, thereby making the inter-packet gap smaller than or greater than the codec sampling interval. Before the voice packets can be played out to the listener or transmitted over a TDM link, this variability in the inter-packet gap must be removed. Removing jitter requires collecting enough voice packets from a channel in a buffer so that the voice packets can be played out with a constant inter-packet gap corresponding to the codec interval. This dejittering operation must be performed individually for each voice call.

QoS CONSIDERATIONS

Traffic management in Asynchronous Transfer Mode (ATM) networks is specified by the ATM forum in the Traffic Management Specification [4]. The TM4.1 specification defines six service categories that are used to provide different levels of QoS guarantees to different types of traffic. Each service category is defined in terms of the characteristics of the traffic that can be afforded this service (called the traffic contract) and the types of QoS guarantees that traffic which conforms to the traffic contract will receive.

The Constant Bit Rate (CBR) service category provides a service similar to that provided by a TDM network. CBR traffic is characterized by a peak cell rate. A conformant CBR traffic stream cannot exceed its peak rate or a maximum cell delay variation. The traffic is guaranteed very low losses and a maximum cell transfer delay. Voice and circuit emulation services are potential users of this service.

The Real-Time Variable Bit Rate (rtVBR) service category provides loss and delay guarantees to traffic whose bit rate is variable. The traffic is characterized by a peak rate, a sustainable rate, and a maximum burst size. A conformant traffic stream must not exceed its sustainable rate over long timescales; it can burst at rates up to its peak rate, up to its maximum burst size. The traffic is guaranteed very low losses and a maximum cell transfer delay. Real-time applications such as voice and video are potential users of this service.

The Non-Real-Time Variable Bit Rate (nrtVBR) service category is intended for non-real-time applications with a bursty traffic pattern. The traffic and conformance criteria for this service are characterized in the same way for the rtVBR service category. The network offers a loss guarantee, but no packet delay guarantees.

The Unspecified Bit Rate (UBR), Available Bit Rate (ABR), and Guaranteed Frame Rate (GFR) service categories are intended for that transport of data traffic. UBR is a best-effort service, where no restrictions are placed on the traffic and no guarantees are provided by the network. ABR provides a loss guarantee and utilizes closed loop feedback control to throttle the traffic sources in order to avoid losses in the network. Finally, GFR is intended to provide a service similar to that offered by frame relay for IP applications.

The TM4.1 specification describes a number of mechanisms for implementing traffic management within the network. Call Admission Control (CAC) is used to determine whether the network has the resources to support a connection and to reserve these resources for the connection. Policing is performed at the edges of the network to make certain that the traffic entering the network conforms to its traffic contract. Shaping is used to transform a traffic stream into one that meets a different traffic contract. Finally, scheduling is used to ensure that the resources reserved by a connection are made available to the cells traversing that connection.

From a traffic management perspective the VoAAL2 application is a user of the network. AAL2 trunks (VCs) are generally established as CBR or rtVBR connections, and the traffic stream produced by the VoAAL2 application for each VC must conform to the traffic contract for that VC. TM4.1 uses the Generic Cell Rate Algorithm (GCRA) for defining the conformance of a traffic stream to its traffic contract. GCRA has two parameters T and τ T is the inverse of the rate allocated to the flow by the network. The rate here could mean either

peak rate or average rate, depending upon the service class. The second parameter τ represents the deviation from the theoretical arrival times of the cells in a flow that can be tolerated by the network. The algorithm maintains the theoretical earliest arrival time for the next cell. When a cell arrives, its actual arrival time is compared to the theoretical arrival time. If the cell has arrived later than the theoretical earliest arrival time or less than τ units of time earlier than this time, then the packet conforms. Otherwise, it does not. The theoretical arrival time is calculated as a function of the actual arrival time of the current cell and the parameter *T*.

VoAAL2 APPLICATION ON IXP2400

We have implemented the Voice over AAL2 (VoAAL2) application on the Intel IXP2400 processor. In this section we describe the design of this application and discuss some of the challenges involved.

DSP to ATM Processing Design

Figure 4 illustrates the major components, data structures and control and data flow for the Digital Signal Processor (DSP) to Asynchronous Transfer Mode (ATM) direction. Thin dotted lines within the figure represent a relationship between data items. Solid lines represent data flow, and thick dashed lines represent control flow.

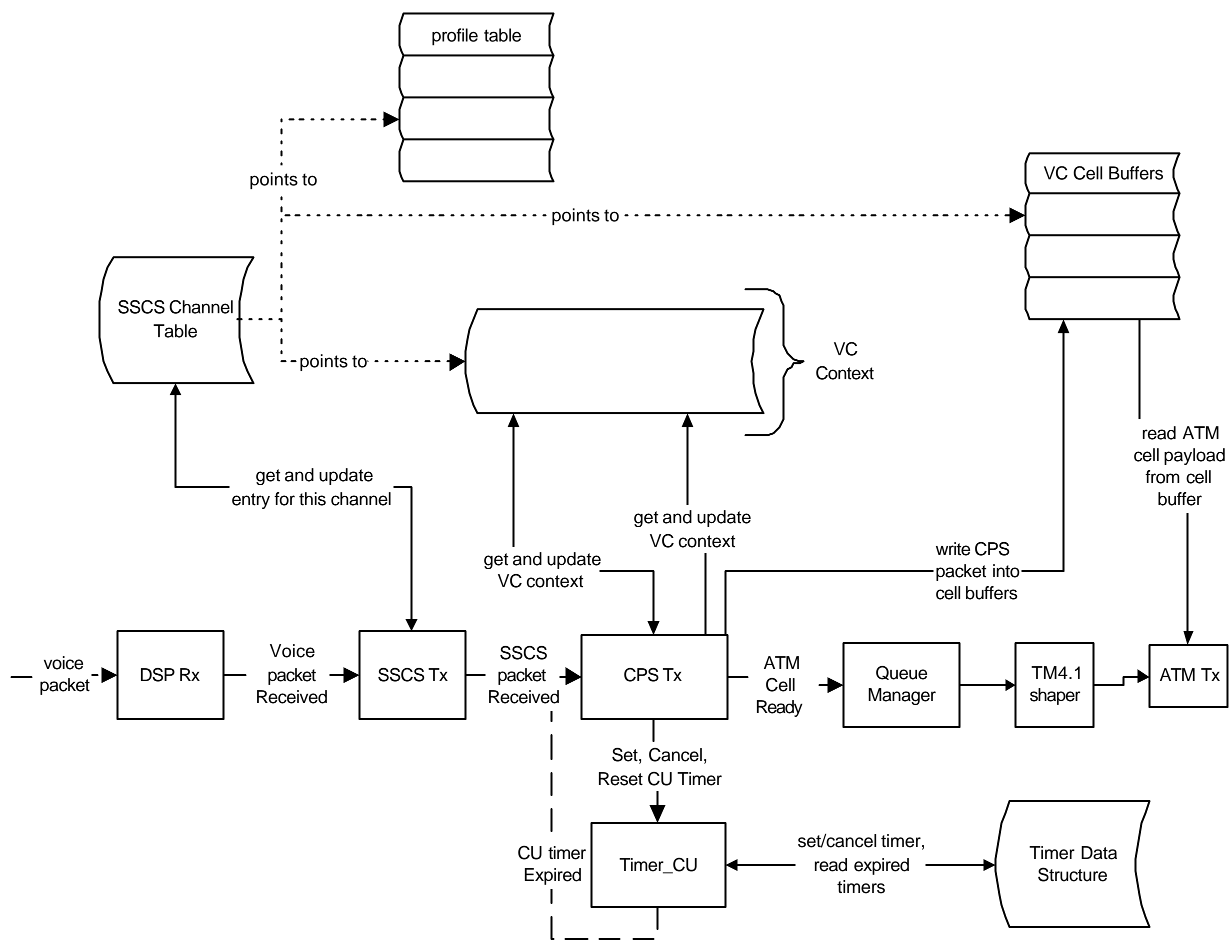


Figure 4: DSP to ATM processing

The **DSP Rx** component receives voice packets from the DSPs. It reads the header that is created by the DSP and extracts the relevant meta-data that describes the voice packet. It then places this meta-data into a message to the SSCS Tx component. It is important to separate out the processing of voice packets from the voice processor because in other versions of this application, the voice packets may come from a different source, such as another AAL2 connection (AAL2 switching), or from a voice-over-IP connection.

The **SSCS Tx** component receives messages from the DSP Rx component, indicating that a voice packet is ready to be processed and performs SSCS processing to produce the value of the User-to-User Indication (UUI) and length fields that go into the Common Part Sublayer (CPS) header. It accesses two main data structures in performing this processing. First, it looks up the entry in the **SSCS channel table** that corresponds to the AAL2 channel over which the voice packet is to be transported. The SSCS channel table contains an entry for each SSCS channel. Each entry contains the Channel Identifier (CID) of this channel, the Virtual Circuit (VC) within which it exists, the profile for this call, and the SSCS sequence number. Next, the SSCS Tx component searches the profile table that describes the profile used for this channel for the entry that correspond to this voice packet. The **profile tables** are read-only tables that describe the UUI field encoding, sequence number interval, and length for each audio encoding algorithm that can be used within the profile. Within a profile table there is an entry for each encoding algorithm supported in the profile. Each voice packet must match one of the entries with the profile. A profile table exists for every profile supported by the application.

When it had completed the SSCS processing, the SSCS Tx component sends a message to the CPS Tx component indicating that a SSCS packet has been received.

The **CPS Tx** component encapsulates SSCS packets in CPS packets and packs CPS packets into AAL2 cells. It is driven by the arrival of two types of messages. Messages indicating that an SSCS packet has been received are sent by the SSCS Tx component. In response to these messages, this component gets the VC context that corresponds to the VC on which the packet has arrived. The **VC context** stores the state of an AAL2 VC. It contains information that is required to determine if the timer for a VC should be set, canceled, reset, or left alone, and information about the cell buffer in which the current cell is being assembled. The CPS Tx component first performs the bookkeeping in order to determine what should be done with the timer_CU for this VC, and sends a message to the timer_CU component indicating the required action. The timer_CU can be set, canceled, reset, or left alone, depending on whether it was previously set and whether the data from the new packet has partially filled a new cell. The CPS Tx component then creates the CPS header to produce a CPS packet that contains an SSCS packet, which contains the voice packet. The CPS packet is written into the current cell buffer immediately following the previous CPS packet destined for this VC. The packet may not entirely fit into the current cell, in which case the part that fits into the current cell is written there and a message is sent to the Queue manager component, indicating that the cell is ready to be sent. The remainder of the CPS packet is written to the next cell buffer. (A maximum size CPS packet can fill up two ATM cells.) The VC context is updated and written back to memory.

Messages indicating that the timer_CU for a VC has expired are sent to the CPS Tx component by the timer_CU component. In response to these messages the CPS Tx component gets the VC context that corresponds to the VC whose timer_CU has expired. It determines how many bytes of padding must be written to complete the cell, writes this padding to the cell buffer, and sends a message to the queue manager component indicating that the cell is ready to be sent. Finally, it updates the VC context to account for the actions that were taken.

The **timer_CU** component implements the timer_CU functionality. It accepts requests to set, cancel, and reset the timer for specific VCs. It is also responsible for firing the individual timer_CUs that are set (and canceled) for individual VCs. The **timer_CU structure** is a calendar queue data structure used to store the timer_CU entries for active VCs. The timers are stored in buckets, and each bucket is associated with a time interval. The timer_CU component wakes up at the end of each time interval and sends timer_CU-expired messages to the CPS Tx component for each VC that had a timer_CU set to go off during the previous time interval.

The **queue manager** component manages a set of queues in SRAM. There is a queue for each ATM VC. Cells are placed into the queue by the CPS Tx component and removed by the TM4.1 shaper component.

The **TM4.1 shaper/scheduler** component consists of three blocks. The TM4.1 shaper block receives input from the queue manager when a cell from a particular VC Queue (VCQ) has been dequeued and there are cells remaining in that VCQ (this is called *cell dequeue without transition*) or when cells are enqueued into an empty VCQ (this is called *enqueue with transition*). The shaper computes the *earliest* departure time for the cell using the Generic Cell Rate Algorithm (GCRA) traffic descriptors. It passes on the *earliest* departure time, the VCQ number, and the service category for the cell onto the TM4.1 writeout block.

The design uses time queues to achieve compliance and provide TM4.1 functionality. Time queues are depicted in Figure 5. The time axis can be divided into small units of cell transmission slots. In each slot, one or no cells depart.

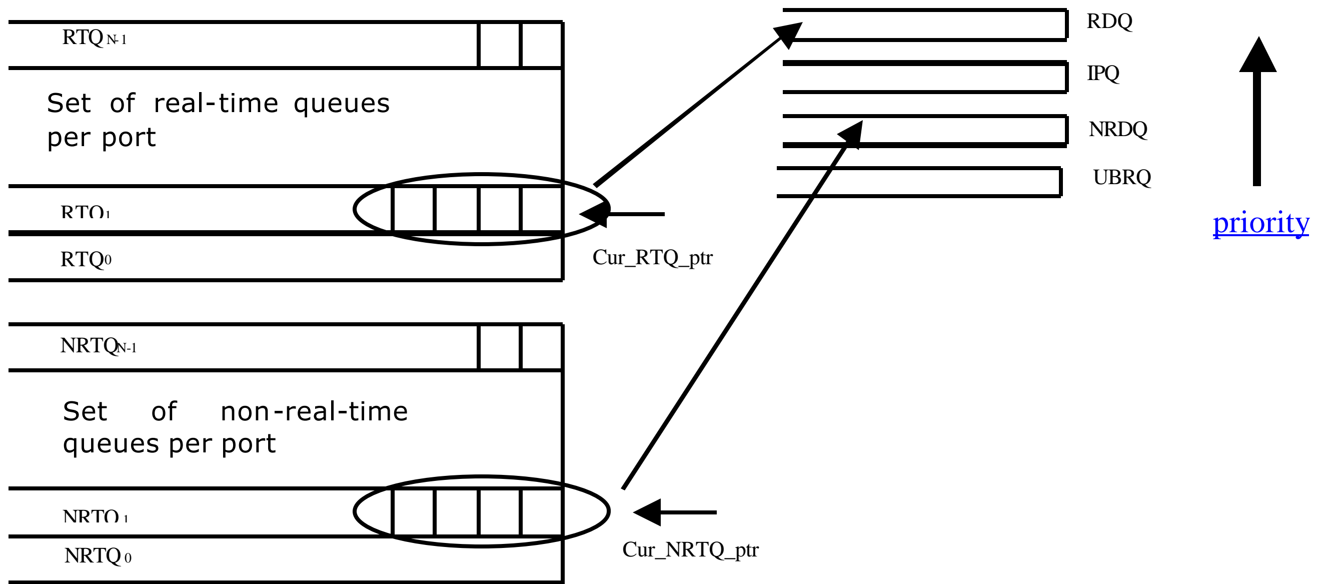


Figure 5: TM4.1 scheduler time queues

A time queue is the aggregation of several cell transmission slots and hence represents an interval of time. The time queue holds the cells that are meant to be transmitted during this time interval. There are a fixed number of time queues in the system (that can be derived based on the link rate), the slowest VC bit rate, and the aggregation level of the time queue. The sum total of all the time intervals represented by the all-time queues would constitute the time horizon. The time horizon is nothing but the time after which the time queues wrap around. There are two sets of time queues: one for real-time traffic (called the *real-time time queue*), such as CBR and rt-VBR, and the other for non-real-time traffic (called the *non-real-time time queue*), such as nrt-VBR and GFR.

The TM4.1 writeout block computes the time queue into which the cell needs to be written based on the *earliest* departure time. Once the time queue is computed for the cell, it writes out the cell into the real-time time queue if the traffic is CBR or rt-VBR and the non-real-time time-queue if the traffic is nrt-VBR. If no space is available in the time queues, the writeout block writes into a different data structure called the Intermediate Priority Queue (IPQ).

The scheduler block schedules out cells from the time queues, IPQ, and the UBR queues. It is essentially a priority scheduler with the highest priority for the real-time time queue, the next priority for IPQ, next for non-real-time time–queue, and the lowest priority for UBR. When scheduling from a time queue, the scheduler is always in sync with or behind the real time. The design ensures that cells are not scheduled ahead of real-time, since this would violate the traffic contracts of the VC and create unfairness in the system.

Finally, the **ATM Tx** component performs the ATM header processing and transmits the cell.

ATM to DSP Design

Figure 6 illustrates the major components, data structures, and control and data flow in the ATM to DSP direction. The **ATM and CPS Rx** component receives ATM cells. When a cell is received, the ATM and CPS Rx component determine the VC to which this cell belongs. The **VC context** for this VC is then read in from SRAM. Since it is possible that a CPS packet was split across cells, this context contains information about such a split CPS packet. The ATM and CPS Rx component reads the contents of the ATM cell into the microengine and steps through the CPS packets contained within. It copies each CPS packet into a packet buffer, maps the Virtual Path Indicator (VPI), Virtual Circuit Indicator (VCI), and CID fields to the channel id, and queues each packet buffer for the SSCS Rx component to process. If a CPS packet is split across ATM cells, the ATM and CPS Rx component stores the reassembly context in the VC context, in order to allow the packet to be completed when the next cell for this VC arrives.

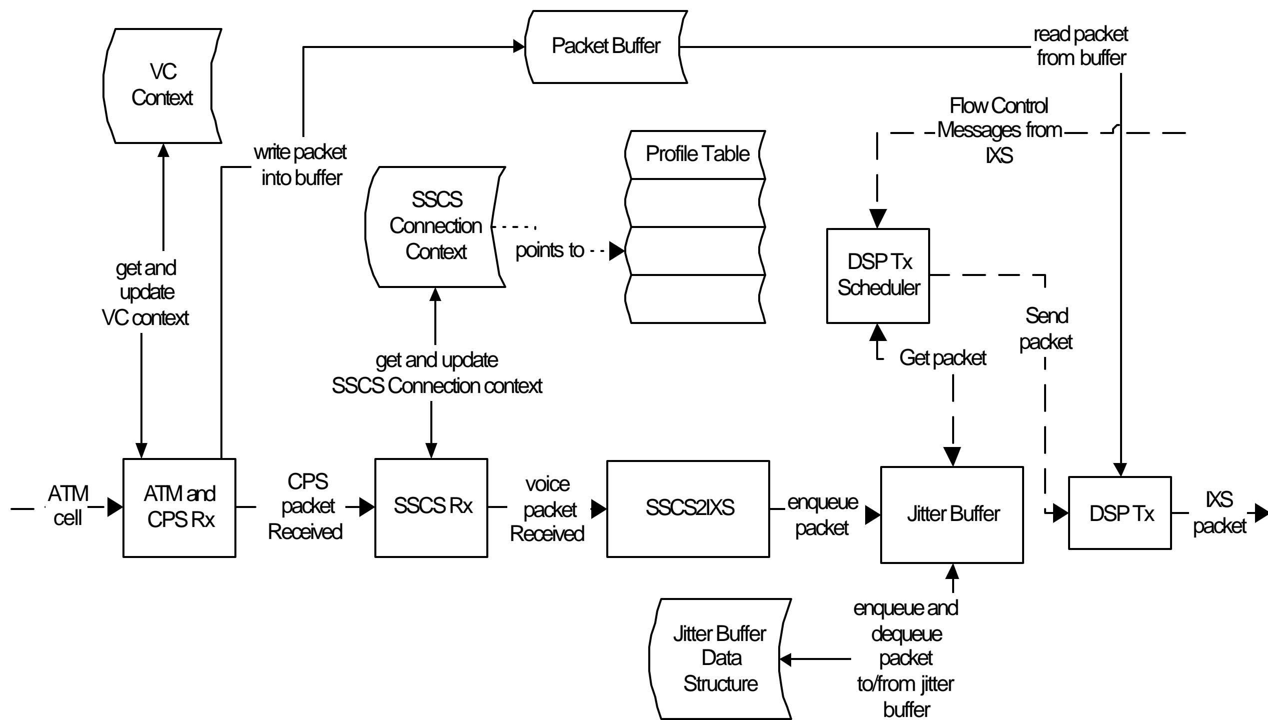


Figure 6: ATM to DSP processing

The **SSCS Rx** component performs the SSCS processing. It uses the channel id to access the **SSCS connection context** for this channel. The context contains the sequence number of the previously received SSCS packet and a pointer to the table that describes the profile for this connection. The structure of the profile tables was described in the previous section. The SSCS Rx component accesses the entry in the profile table that corresponds to the UUI field and length of the SSCS packet (in the CPS header). This entry specifies the encoding algorithm that was used, along with information used to determine the expected sequence number and corresponding timestamp for this packet. SSCS Rx checks the sequence number against the one received in the packet and generates the timestamp. If there are no errors, the packet is passed to the SSCS2DSP component.

The **SSCS2DSP** component creates a DSP packet header from the information that the ATM and CPS Rx and SSCS Rx components have extracted from the packet and profile table. It then passes the packet to the jitter buffer component.

The **jitter buffer** component enqueues the packet into a per-channel queue. The purpose of the Jitter Buffer component is to eliminate some of the jitter introduced into the voice packet stream in the ATM network. It does this by placing packets into proper time sequential order, applying a specified jitter delay, and playing them back at the proper rate (with jitter removed).

The **DSP Tx scheduler** component is responsible for scheduling the transmission of DSP packets to the DSP. This component registers itself with the MSF in order to receive flow control messages from the DSP. In each flow control message, the DSP indicates one or more channels, on which it is ready to receive a packet. For each such channel, the DSP Tx scheduler component asks the jitter buffer component to dequeue a packet from the jitter buffer. The jitter buffer returns either a buffer handle or an error. If the jitter buffer returns a buffer handle, the DSP Tx scheduler component passes this handle to the DSP Tx component. If the jitter buffer component returns an error, the DSP Tx scheduler component creates a DSP packet that indicates silence (an SID) and passes it to the DSP Tx.

The jitter buffer component receives requests from the DSP Tx scheduler to dequeue a packet on a specific channel. It determines if a packet is queued and ready to send for that channel and responds with the buffer handle of that packet or an error.

Finally, the **DSP Tx** component transmits voice packets to the voice processors.

CHALLENGES AND LESSONS LEARNED

In this section we discuss some of the challenges that must be surmounted in developing a Voice over AAL2 (VoAAL2 application) on a Network Processor Unit (NPU).

Processing Asynchronous Inputs within Many Contexts

The VoAAL2 application must support a large number of AAL2 Virtual Circuits (VCs). There is a requirement that within each VC, cells/packets be processed in the order in which they were received. (In the DSP to ATM direction, voice packets destined for a VC should be processed in order, while in the ATM to DSP direction, ATM cells received on a VC must be processed in order.) The data rates within VCs can be fairly small, so at any given moment the application is holding/processing cells/packets from a small subset of this total number of VCs. Because there are many VCs, the packets that the application is processing/holding at a given time are most likely all from different VCs, although this is not guaranteed and cannot be assumed by the application. The challenge is how to serialize the processing of cells/packets within each VC, while allowing cells/packets from different VCs to be processed in parallel.

Another problem is that the CPS Tx component must process voice packets received by the system as well as react to the expiration of the timer_CU. Timer_CU expiration events are not regular or predictable, since they are a function of the traffic patterns on individual VCs. The amount of processing required to process a packet that has arrived is much larger than that required to react to the expiration of the timer_CU.

We solve the problem of having to serialize the processing of packets/cells within each VC by dynamically binding VCs to threads. A thread receives a packet or message, determines which VC it belongs to, checks to see if any other thread is already processing packets/messages for that VC, and locks the VC if it is not already locked by another thread. The thread then processes the packet or message. When it has completed processing the packet or message, it checks to see if any other packets or messages have been queued for it to process (associated with this same VC). The thread processes any packets or messages that have been queued, and when there is none left, it unlocks the VC. On the other hand, if another thread has already locked the VC, the packet or message is queued for this other thread to process.

The process of locking a VC, unlocking a VC, and checking to determine if a VC is locked must be performed in an atomic fashion in order to ensure that two threads do not lock the same VC. In our design the entire component is implemented within a single ME, so we use a built-in CAM for storing the identity of the VC that is locked by each thread, allowing the operations of locking, unlocking, and checking to see if a VC is locked to be performed in one operation .

We found that the IXP2400 provides good support for the asynchronous programming model used in the VoAAL2 application. Central to this support are the CAM and local memory that are included in each ME. The IXP2400 also provides the basic support required to distributed this type of processing across multiple MEs. It provides atomic test and set operations in the shared SRAM, which can be used to implement locks. However, SRAM operations have a fairly large latency, making it difficult to use this mechanism for locking in high-performance applications. Additional hardware support for performing distributed locking from threads on multiple MEs would make it easier to implement such multi-ME asynchronous applications. On the other hand, it is generally possible to partition an application into components in such a way as to avoid asynchronous components that run on more than one ME.

Bit- and Byte-Level Memory Access

The CPS header and AAL2 cell are tightly packed structures, where fields are not aligned on four-, eight-, or even one-byte boundaries. This means that the VoAAL2 application must read and write from/to arbitrary bit and byte addresses as it creates/parses CPS packet headers and packs/unpacks CPS packets from AAL2 cells. This presents a challenge for any processor because memory systems generally support reads/writes of four- or eight-byte chunks of data, addressed on four- or eight-byte boundaries.

Our implementation utilizes specialized byte alignment hardware of the IXP2400 processor to merge and align the CPS packets as we pack/unpack them into/from AAL2 cells. Bit fields are accessed utilizing mask and shift operations provided by the Arithmetic Logic Unit (ALU). Efficient support for such data access is critical to support applications such as VoAAL2, where packets are small and protocol overhead must be minimized.

The problem of having to access unaligned data is solved by some combination of providing specialized hardware instructions and simply providing sufficient processor speed to allow the applications to perform the required data manipulations within the required time budget. We found that the IXP2400 provides a reasonable combination

of processing speed and specialized instructions to support this application.

Jitter Buffer

The purpose of the jitter buffer is to receive voice packets, place them in proper time sequential order, provide a specified jitter delay, and then present them for transmission. The jitter buffer must be large enough so that the slowest packets can arrive in time to be played out in the correct sequence. On the other hand, the jitter buffer must be small enough such that the delay introduced is minimized. In order to address these conflicting requirements, the jitter buffer can be dynamically resized based on measurements of actual network jitter. On lightly loaded paths, this allows for a minimum jitter delay and a higher quality of speech with less noticeable turnaround delay. On congested paths, the jitter delay can be increased so that fewer packets are missed or dropped due to the irregularity of their timing but with a more noticeable turnaround delay.

Implementing a jitter buffer offers some new challenges when compared to traditional First In First Out (FIFO) queues. The jitter buffer is a sorted queue based on the timestamps of arriving voice packets. Therefore, packets can be inserted in the middle of the jitter buffer. Packets can be dropped from a jitter buffer for two reasons: 1) the buffer is full; or 2) the packet is received too late. When packets are dropped because the queue is full, they are dropped from the front of the queue (packets with the oldest timestamp are dropped). Because the queue is allowed to contain packets representing a fixed time interval (the jitter delay value), the arrival of one voice packet may cause multiple older voice packets to be dropped if the time difference between the newest packet's timestamp and the oldest packet's timestamp is greater than the jitter delay value. Packets with duplicate timestamps are dropped. A further challenge is that a separate queue must be maintained for each of the many thousands of voice channels that are handled by the application.

Our implementation uses circular queues to implement the jitter buffer. Each circular queue has pointers to the packets with the oldest and newest timestamps. The position into which a new packet is inserted is a function of the difference between the packet's timestamp and the oldest packet's timestamp, along with the codec interval. To calculate this position we need to divide the timestamp difference by the coded interval. We implement this using fast reciprocal multiplication utilizing the multiplier of the IXP2400 MEs. Once the position is calculated, the insertion of the packet into the jitter buffer is the same as an insert into an array of the order O(1).

The circular queues may have "holes," positions with no packets. When a packet must be removed from the jitter buffer, it is necessary to quickly skip over the holes to get to the position with a valid packet. We implement this search for a valid packet in O(1) time by making use of the Find First Bit Set (FFS) instruction provided by the IXP2400 MEs. By maintaining a bit-mask of positions in the circular queue with valid packets, and using the FFS instruction, we can remove the next valid packet from the jitter buffer in constant time.

In summary, the jitter buffer implementation takes advantage of the hardware features provided by the IXP2400 network processor to implement, insert, and remove operations in O(1) time. This allows for an efficient jitter buffer implementation that scales to a large number of voice channels.

TM4.1 Real-Time Scheduler

There are many challenges that must be overcome in developing a TM4.1-compliant scheduler. TM4.1 requires per-VC shaping and scheduling, and the number of VCs can be very large. Also, the implementation must scale with increases in line rate, as well as numbers of VCs.

The most interesting challenge is in providing the real-time scheduling required to support the CBR and rtVBR service classes. When servicing CBR and rtVBR traffic, the packet scheduler must transmit each cell within a certain time window in order for it to conform to the traffic contract.

Because the IXP2400 performs non-preemptive Round Robin scheduling, it is difficult for the software to perform such real-time scheduling. When a thread sets a timer and goes to sleep, expecting to be awakened when the timer has expired, it cannot be guaranteed that it will get awakened the instant that the timer expires, because another thread might be executing at the time that the timer expires, and other threads might be ahead of this thread in the Round Robin schedule. We found that TM4.1 scheduling is still possible, although the software must be carefully tuned to place an upper bound on the time between the expiration of a timer and the time that the thread is awakened, and the schedule must take this bound into account when setting its timers.

CONCLUSION

The flexibility and programmability of next-generation Network Processor Units (NPUs) will make them a key component of next-generation telecommunications equipment. NPUs can support a variety of packet-processing applications, with a variety of different requirements. The Voice over AAL2 (VoAAL2) application that we discussed in this paper presents a

number of challenges. We have demonstrated that these challenges can be overcome and that applications such as VoAAL2, with strict Quality of Service (QoS) requirements and asynchronous inputs, can be performed on an NPU.

The VoAAL2 application is most naturally implemented using an asynchronous programming model. We found that the IXP2400 naturally supports such a programming model. Support for asynchronous components that span MEs could be improved by adding support for a distributed lock manager.

The AAL2 application requires a lot of bit- and byte-level data access. The IXP2400 provides all of the necessary facilities to perform these operations while meeting the performance requirements of the application. Finally, the VoAAL2 application requires real-time scheduling to conform to the TM4.1 traffic contract. Although the IXP2400 does not support preemptive scheduling, the software can be tuned to perform the required scheduling in conformance with TM4.1.

REFERENCES

[1] S. Lakshmanamurthy, et. al, "Network Processor Performance Analysis Methodology," *Intel Technology Journal*, Vol. 6 issue 3, August 2002.

[2] ITU-T Recommendation I.363.2, Series I: B-ISDN ATM Adaptation Layer Specification: Type 2 AAL, ITU, Geneva, Switzerland, 1997.

[3] ITU-T Recommendation I.366.2, AAL Type 2 Service Specific Convergence Sublayer for Trunking ITU, Geneva, Switzerland, 1999.

[4] The ATM Forum, "Traffic Management Specification Version 4.1," af-tm-0121.000, April 1999.

[5] F. Baker, "RFC 1812 Requirements for IP Version 4 Routers," *IETF*, June 1995.

AUTHORS' BIOGRAPHIES

Jaroslaw J. Sydir is a network architect in the Silicon Development Group of the Network Processor Division at Intel Corporation. His interests are in the areas of signaling protocols, traffic management, and distributed real-time systems. He received his B.S. in Computer Engineering from Case Western Reserve University in 1988, and an M.S. degree in Systems Engineering from Case Western Reserve University in 1989. He can be reached at jerry.sydir@intel.com.

Prashant Chandra is a senior staff network architect in the Software and Systems Engineering group of the Network Processor Division at Intel Corporation. His interests are in the areas of programmable networks, signaling protocols, and traffic management. He received his B.E. degree in Electronics Engineering from Bangalore University in 1991, an M.S. degree in Computer Engineering from West Virginia University in 1994, and a Ph.D. degree in Computer Engineering from Carnegie Mellon University in 2000. He can be reached at prashant.chandra@intel.com.

Alok Kumar is a staff software architect in the Software and Systems Engineering group of the Network Processor Division at Intel Corporation. His interests are in the areas of high-speed programmable routers, quality of service, and computer graphics. He received his B.Tech degree in Computer Science from the Indian Institute of Technology, Delhi, in 1999, and his M.S. degree in Computer Science from the University of Texas at Austin in 2001. He can be reached at alok.kumar@intel.com.

Sridhar Lakshmanamurthy is a senior staff architect in the Silicon Development Group of the Processor Division at Intel Corporation, focusing on understanding edge/access networking applications, analyzing the performance of Intel's network processor solutions, and defining future enhancements to these solutions. Prior to joining the Network Processor Division, Sridhar focused on platform performance analysis for Intel server chipsets for Xeon™ & Itanium® Product Family (IPF) processors, and system bus specifications for the IPF processors. Sridhar joined Intel in 1993 after receiving an M.S. degree in Computer Engineering from Rice University in Houston, Texas. He can be reached at sridhar.lakshmanamurthy@intel.com.

Longsong Lin is the senior staff network architect in the NPG technology office. He was the principal system architecture at AMCC, CTO at Opix Networks, senior research staff and architect at NEC, Japan, scientist at Swiss Federal Institute of Technology, Switzerland, VP of Engineering at Jato International Inc., professor and chairman at National Yunlin University of Science and Technology, Taiwan, visiting professor at the University of Illinois, Urbana Champaign, visiting scholar at Purdue university, and a member of the Taiwan Public TV Organizing Committee. He received his Ph.D. degree and M.S. degree in Electrical Engineering at Purdue University. He is currently involved with several projects at Intel, including modular system platforms and next-generation network processor architecture, as well as business and technology developments in APAC. He can be reached at longsong.lin@intel.com.

Muthu Venkatachalam is a network architect in the Software and Systems Engineering group of the Network

Processor Division at Intel Corporation. His interests lie in network processor architecture and programming, QoS algorithms, traffic management, systems modeling and analysis, and keeping pace with the innovations in today's networking industry. He can be reached at muthaiah.venkatachalam@intel.com.

Xeon™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries. Itanium® is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Copyright © Intel Corporation 2002. This publication was downloaded from http://developer.intel.com/

Legal notices at http://developer.intel.com/sites/developer/tradmarx.htm.

For further information visit:

developer.intel.com/technology/itj/index.htm

Copyright © 2002, Intel Corporation. All rights reserved.
Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.
For a complete listing of trademark information vistit: www.intel.com/sites/corporate/tradmarx.htm