ELSEVIER

Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE @ DIRECT®

Computer Networks 41 (2003) 563–586

COMPUTER NETWORKS

www.elsevier.com/locate/comnet

# A highly flexible, distributed multiprocessor architecture for network processing

Muthu Venkatachalam *, Prashant Chandra, Raj Yavatkar

*Intel Communications Group, Intel Corporation, JF3-462, 2111 NE 25th Ave, Hillsboro, OR 97124, USA*

**Abstract**

Network processors (NPs) are an emerging field of programmable processors that are optimized to implement data plane packet processing networking functions. Unlike the general-purpose CPUs that rely heavily on caching for improving performance, the lack of locality in packet processing and need for high-performance I/O have forced designers to come up with innovative architectures that can hide memory latency while still processing packets at high data rates. Most of these NPs use some type of multiprocessing in combination with a hierarchy of memory types to achieve high performance. In addition, to keep up with packets arriving at high data rates over multiple incoming media interfaces, an NP must perform fast I/O and memory operations such as packet storage, table lookup, and extraction of fields in packet headers. We describe an architecture that uses a combination of distributed memory architecture and one or more multithreaded processors to achieve the necessary performance. We describe the challenges in programming such a processor including the issues related to consistency and maintaining packet ordering. We also present a programming model for generic network applications that uses software pipelines. We then demonstrate the use of the programming model in implementing two applications, namely, mapping traffic management algorithms onto a multithreaded architecture and an implementation of a media gateway based on voice-over-AAL2.
© 2003 Elsevier Science B.V. All rights reserved.

*Keywords:* ATM; AAL2; IXP; Media gateway; Media signal processor; Network processor; Programming; Pipelining; Traffic management

## 1. Introduction

Network processors (NPs) are an emerging class of programmable processors used as a building block for implementing packet processing applications in networking systems such as switches and routers. They are highly optimized for fast packet processing and I/O operations. They are typically characterized by distributed, multiprocessor, multithreaded architectures designed for hiding memory latencies in order to scale up to very high data rates. They tend to have multiple fast path forwarding engines with access to a slow path exception handler. With all these characteristics, NPs usually pose interesting challenges to programmers.

In this paper, we provide an overview of the hardware architecture for Intel's second generation NPs. We describe the challenges involved in programming these processors. We then present an

* Corresponding author.
*E-mail addresses:* muthaiah.venkatachalam@intel.com (M. Venkatachalam), prashant.chandra@intel.com (P. Chandra), raj.yavatkar@intel.com (R. Yavatkar).

1389-1286/03/$ - see front matter © 2003 Elsevier Science B.V. All rights reserved.
doi:10.1016/S1389-1286(02)00450-4

overview of a programming model for implementing networking applications on these processors. Finally, we show how the NPs can be used to solve practical problems, by describing an implementation of ATM based traffic management algorithms and an ATM AAL2-based media gateway application using the programming model.

The rest of the paper is organized as follows. Section 2 provides an overview and motivation for the IXP hardware architecture. In Section 3, we describe the challenges involved in implementing networking applications on a distributed, multiprocessor architecture of the IXPs. In Section 4, we then describe a suitable programming model designed to meet such challenges. Section 5 describes how a voice-over-ATM application is implemented for a media gateway and Section 6 provides the details of an implementation of the ATM traffic management algorithms.

## 2. Hardware architecture overview

In this section, we first provide an overview of the architecture of Intel network processors with the IXP2400 as an example.

### 2.1. IXP2400 architecture

The overall architecture of an IXP2400 is shown in Fig. 1 [3].

#### 2.1.1. Motivation

The two major parts of the IXP architecture are the Intel Xscale® processor core (referred to as the Xscale® core) and a number of RISC processing engines called microengines (MEs). The network processing unit (NPU) architecture is motivated by the need to provide a silicon building block for packet processing. In any packet processing application, two types of operations must be performed, namely, so-called *fast-path* operations and *slow-path* operations. The fast-path operation is performed typically on each packet and must be completed as quickly as possible to keep up with the line rate. An example of a fast-path operation includes IPv4 forwarding which involves looking up the destination IP address in a forwarding table to determine the outgoing interface over which an IP datagram must be sent, updating the IPv4 protocol header, and then sending out the packet on the specified outgoing interface. On the other hand, a slow-path operation is needed only in an

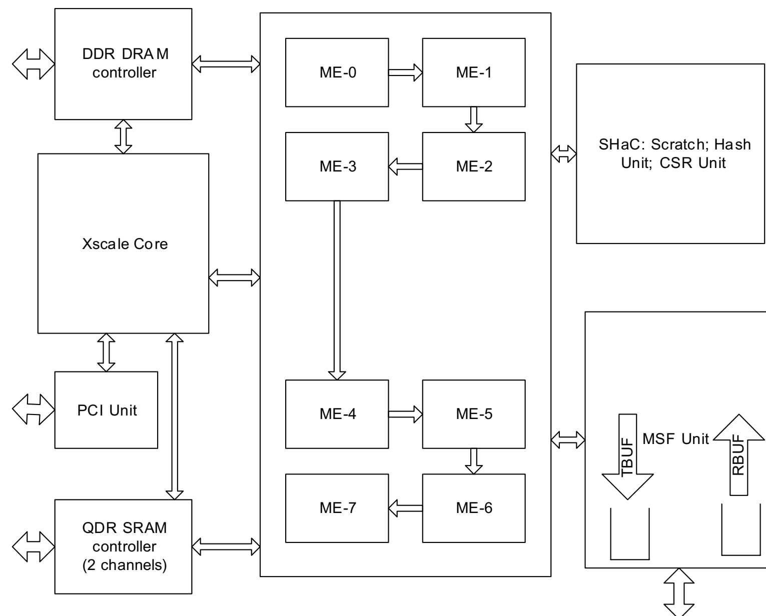


Fig. 1. The block diagram of an IXP 2400.

exceptional case and takes much longer to perform than a fast-path operation. An example of a slow-path operation occurs when a network device receives an IPv4 datagram with certain header options turned on. This happens rarely and processing header options requires a lot more time than the usual fast-path operation.

Apart from the packet processing operations, every network device or line card needs to execute some control plane protocols such as routing protocols and signaling protocols. These protocols run in the background and must process incoming control packets such as routing protocol updates or signaling requests for call establishment.

The IXP architecture's two main building blocks are designed to support both packet processing and control protocol processing. The Xscale® core is designed for slow-path processing and control protocol processing whereas the MEs are designed for fast-path packet processing.

*2.1.2. Slow-path and control plane processing*

Each IXP includes an XScale™ core. The Intel® XScale™ core is a general purpose 32-bit RISC processor (ARM* Version 5 Architecture compliant) used to initialize and manage the NP, and used for exception handling, slow-path processing and other control plane tasks. The Intel® XScale™ microarchitecture incorporates an extensive list of architecture features that allows it to achieve high performance. The Intel® XScale™ microarchitecture implements a 32-Kbyte, 32-way set associative instruction cache with a line size of 32 bytes. A mechanism to lock critical code within the cache is also provided. The microarchitecture also implements a 32-Kbyte, 32-way set associative data cache and a 2-Kbyte, 2-way set associative mini-data cache. Each cache has a line size of 32 bytes, and supports write-through or write-back caching.

*2.1.3. Fast-path processing*

The fast-path packet processing part of the architecture is motivated by the following observations:

- *Multithreaded MEs*: As processing speeds have increased, memory access speeds and latencies have not kept pace. When an NPU must process an incoming stream of packets at line rate, it must complete processing a packet within a short amount of time. For example, at OC-48 line rate, a minimum size POS packet arrives approximately every 100 cycles. However, even simple processing requires some lookup operations and other memory accesses. With a SRAM access latency estimate of roughly 80–100 cycles and SDRAM access latency estimate of 150–200 cycles, it is not possible to complete processing a packet before the next one arrives. The IXP architecture uses multiple processing engines (microengines) where each ME has eight hardware threads. Assigning each packet to a separate thread on one or more MEs allows processing of multiple packets in parallel. Thus, the first packet does not have to finish processing in 100 cycles as long as there are enough threads to process subsequent packets before the first packet is completely processed. The IXP2400 network processor has eight MEs working in parallel on the fast packet-processing path. Both the Xscale core and the MEs run at 600 MHz clock frequency.
- *Distributed, shared memory hierarchy*: Packet processing typically involves table lookups to match fields in the packet header against values stored in lookup tables and also involves buffer management as packets are stored first and then forwarded after they undergo processing. The processor has distributed, shared memory architecture. For example, it supports two types of external memories, QDR SRAM for low-latency access to smaller data structures used in lookup operations and DDR SDRAM for larger storage needed for packet buffers and bulk data transfers. Both SRAM and SDRAM address spaces are shared among all MEs and the data in those memories is accessible to each ME thread on an equal basis. In addition, the processor includes an on-chip SRAM (16 KB) that is shared among all the MEs and a small amount (640 32-bit words) of local memory per ME. The on-chip SRAM is designed to provide fast access to the processing state, packet headers, or meta-data shared among the code running on different MEs whereas the local

memory in ME is useful to cache data needed by multiple threads within a ME.

- *Special-purpose units*: MEs are fully programmable, general-purpose engines and can be programmed to implement any, arbitrary packet processing (or other) functions. However, packet processing functions involve some well-defined, commonly needed functions such as hashing, CRC, crypto, etc. To avoid implementing such standard functions from scratch, the IXP includes on-chip, special-purpose hardware units for hashing and CRC computation. In addition, a variant of IXP includes DES, 3-DES [7], and AES [6] blocks for bulk encryption and authentication functions.
- *Built-in media interfaces*: The processor has a flexible 32-bit media switch interface configurable as $1 \times 32$-bit or $2 \times 16$-bit or $4 \times 8$ bit or $1 \times 16 + 2 \times 8$ bit interfaces. Each interface is configurable as media standard SPI-3 or CSIX-L1 or Utopia 1/2/3 interfaces.

### 2.2. Microengine

The MEs do most of the per-packet processing in an IXP. In IXP 2400, there are eight MEs, connected in two clusters of four ME as shown in Fig. 1. The MEs have access to all shared resources (SRAM, DRAM, MSF, etc.) as well as private connections between adjacent MEs (referred to as "next neighbors"). The next-neighbor connections are only one-way as shown in Fig. 1 creating a pipeline of MEs (one through eight) in terms of nearest-neighbor communication.

The ME provides support for software controlled multithreaded operation. Given the disparity in processor cycle times vs. external memory times, a single thread of execution will often block waiting for the memory operation to complete. Having multiple threads available allows for threads to interleave operation—there is often at least one thread ready to run while others are blocked. This improves the usage of the ME resources.

*Control store*

The control store is a RAM, which holds the program that the ME executes. It holds 4096 instructions, each of which is 40-bits wide. It is initialized by the Intel® XScale™ core.

*Contexts*

There are eight hardware contexts (or threads) available in each ME. To allow for efficient context swapping, each context has its own register set, program counter, and context specific local registers. Having a copy per context eliminates the need to move context-specific information to/from shared memory and ME registers for each context swap. Fast context swapping allows a context to do computation while other contexts wait for I/O (typically external memory accesses) to complete or for a signal from another context or hardware unit.

*Data path registers*

Each ME contains four types of 32-bit data path registers:

1. 256 general purpose registers (GPRs),
2. 512 transfer registers,
3. 128 next neighbor (NN) registers,
4. 640 32-bit words of local memory (LM).

GPRs are used for general programming purposes. They are read and written exclusively under program control. GPRs, when used as a source in an instruction, supply operands to the execution data path. When used as a destination in an instruction, they are written with the result of the execution data path. The specific GPRs selected are encoded in the instruction.

Transfer registers (abbreviated Xfer registers) are used for transferring data to and from the ME and locations external to the ME (for example DRAM, SRAM, etc.). There are four types of transfer registers. Typically, the external units access the transfer registers in response to instructions executed by the MEs. However, it is possible for an external unit to access a given ME's transfer registers either autonomously, or under control of a different ME, or the Intel® XScale™ core, etc.

Next neighbor (NN) registers, when used as a source in an instruction, supply operands to the execution data path. NN registers in an ME can be written either by an adjacent ME or the same ME.

This is a fast, optimal way of communication between the two MEs.

*Special hardware features in the ME*

The ME also provides the following special hardware blocks to assist in various packet-processing tasks:

- CRC unit—compute 16b and 32b CRC. This accelerates the performance of ATM AAL5 SAR applications.
- Pseudo-random number generator—assists in supporting QoS algorithms for congestion avoidance, e.g., WRED [8] and RED.
- Time-stamp, timer—assists in supporting metering, policing, rate shaping functionality required in IP DiffServ and ATM traffic management services.
- Multiply unit—assists in QoS blocks such as policing, congestion avoidance.
- CAM—each ME includes a 16-entry CAM (content addressable memory) that is used to cache per-connection state that is shared among multiple instances of a function running in parallel on threads within an ME.

### 2.3. DDR DRAM

IXP2400 supports a single 64-bit channel (72 bit with ECC) of DRAM. DRAM sizes of 64, 128, 256, 512 MB and 1 GB are supported. An address space of 2 GB is allocated to DRAM. The memory space is guaranteed to be contiguous from a software perspective. If less than 2 GB of memory is present, the upper part of the address space is aliased into the lower part of the address space and should not be used by software.

### 2.4. QDR SRAM

The IXP2400 network processor has two independent SRAM controllers, each of which support pipelined QDR synchronous static RAM (SRAM) and/or a coprocessor that adheres to QDR signaling. Any or all controllers can be left unpopulated if the application does not need to use them. SRAM is accessible by the MEs, the Intel® XScale™ core, and the PCI Unit (external bus masters and DMA).

*Queue data structure commands*

The ability to enqueue and dequeue data buffers at a fast rate is key to meeting line-rate performance. This is a difficult problem as it involves dependent memory references that must be turned around very quickly. The SRAM controller includes a data structure (called the Q_array) and associated control logic in order to perform efficient enqueue and dequeue operations. The Q_array has 64 entries, each of which can be used in one of the following ways:

- Linked-list queue descriptor.
- Cache of recently used linked-list queue descriptors (the backing store for the cache is in SRAM).
- Ring descriptor.

### 2.5. Scratchpad and hash unit

The scratchpad unit provides 16 KB of on-chip SRAM memory that can be used for general-purpose operations by the Intel® XScale™ core and the ME. The scratch also provides 16 hardware rings that can be used for communication between MEs and the core.

The hash unit provides a polynomial hash accelerator. The Intel® XScale™ core and MEs can use it to offload hash calculations in applications such as ATM VC/VP lookup and IP 5-tuple classification.

## 3. Programming challenges

Network processing applications are targeted at specific data rates. In order to meet these throughput requirements, a NP must complete the packet processing tasks on a given packet before another packet arrives. The absolute worst-case throughput requirements would typically be for minimum sized packets arriving back-to-back. For example, at OC-48 (or 2.4 Gbps) rate, a minimum size packet (46 bytes) will arrive every 100 cycles. To keep up with the back-to-back arrival of minimum size packets at the line rate, the NPU must

complete processing the packet in 100 cycles. Such a constraint and potential solutions to meet the constraint pose some interesting programming challenges:

- *Achieving a deterministic bound on packet processing operations*: The constraint to keep up with the back-to-back packets arriving at line rate affects the programming significantly since the time to process a packet on each ME would now be the time it takes to transmit the minimum sized packet on the wire. This means we need to design the software in such a way that the number of clock cycles to process the packet on each ME does not exceed an upper bound. This is particularly hard for applications like schedulers, where we may have a lot of queues in the system and a search for the queue that is eligible for scheduling needs to be performed. This search could be extremely variable if one has to walk through all the queues in a linear fashion. We will not be able to meet the line rates in such a case. Hence it is important that the software uses the right kind of data structures and is designed in such a way that it meets the targeted upper bound on the processing cycles per packet.
- *Masking memory or I/O latency through multithreading*: Even if the data structures and processing blocks are designed to complete processing within a definite time interval, it is not sufficient to meet the line-rate processing requirements because a programmer must still deal with memory and I/O latencies that are much higher than the amount of processing budget available to each packet. Therefore, another important challenge in programming a NP is to utilize the multiple hardware threads effectively to mask the I/O latencies. Several techniques such as pipelining and multiprocessing can be employed to effectively hide the I/O latencies as explained in the next section.
- *Ensuring consistency of shared data*: Once one uses multithreading to process multiple packets in parallel, it is important to maintain consistency of the updates to shared data. This can be achieved by using locks in memory or by ensuring that the threads operate in strict order on a timeline making sure at most one thread accesses shared data at any time. Use of locks is easier but tends to be very slow whereas strict thread ordering avoids the latencies involved in memory-based locking. However, the latter approach suffers from a drawback that the rate at which an ME operates is limited by the thread that requires the longest time to process the packet.
- *Maintaining packet ordering in spite of parallel processing*: The other significant challenge in programming the NPs is to maintain packet ordering. This is extremely critical for applications like media gateways and traffic management, where we should not re-order packets on a voice call and cells on a VC. Packet ordering can be guaranteed using a couple of techniques:
  - Using sequence numbers for packets being processed.
  - Use strict thread ordering: If we make sure that threads operate in a strict order, then packets are assigned to threads in order of their arrival and, therefore, the packet will complete processing and will go out in order.

Thus, a combination of processing of packets using multiple threads running in parallel and strict thread ordering can meet the challenges described above. A programming model based on this approach is described in the next section.

## 4. Microengine programming model

Fast path packet processing functions are implemented on the IXP2400 microengines. The MEs provide hardware support for up to eight threads and present a RISC instruction set that is optimized for packet processing tasks. A variety of programming models can be employed on the MEs. The choice of a programming model depends mainly on the performance requirements of the application. This section explores a few common programming models and evaluates the pros and cons of each model.

The first step in developing the application is to break down the application into a series of packet

processing tasks that can be run in parallel. The second step is to determine how these tasks can be allocated to the array of MEs taking into account the performance requirement for each task. Suppose that, for a given packet processing application, the worst case inter-arrival time of packets is represented by $T_a$. The worst-case inter-arrival time corresponds to the maximum rate at which packets can be expected to arrive for the given application. In most cases, this corresponds directly to the arrival rate for minimum sized packets.

### 4.1. Multithreading

The most common method of using an ME is to perform the same task or set of tasks in parallel on different packets. The hardware support for multithreading in an ME, allows the processing of up to eight packets in parallel. Fig. 2 shows how multithreading is employed to process packets in parallel. Each packet is assigned to one thread which executes the packet processing task on that packet. When a thread swaps out waiting for any I/O operation to complete (e.g., reading state from external memory), the next thread executes the same task on the next packet.

If a task is allocated to a single ME, then the granularity of a task should be such that the worst-case performance requirements for that task "fits" within the capacity of the ME. If $f$ is the ME clock frequency, $T_a$ is the worst-case performance requirement for the task, and $I$ is the number of instructions that must be executed for that task, then the task fits within the capacity of the ME if $T_a > I/f$. Because eight packets are processed in parallel, each thread can take up to $8 \times T_a$ cycles to complete the processing task on a packet. This time includes both the active execution time and the time spent waiting for I/O operations to complete. However, because all eight threads share the ME processor, each thread can execute for at most $T_a$ time on any one packet.

One important consideration is to maintain packet ordering in the presence of parallel processing of packets. Packet ordering can be maintained using two different approaches. In the first approach, threads are allocated packets in order and are also forced to execute in the same order. This approach is shown in Fig. 2. In this case, the packet order is maintained as a byproduct of ordered thread execution. In the second approach, threads are allowed to run asynchronously. But packet order is maintained by using sequence numbers. Packets are assigned sequence numbers upon entry into an ME and exit the ME in the order of increasing sequence numbers.

When the execution time for a fast path packet processing application is greater than the capacity of a single ME, multiple MEs must be used in parallel to implement that application. A given set of tasks can be assigned to MEs in two different ways. The first way is to construct a pipeline of MEs with each ME executing a particular task. The second way is to use the MEs in a multiprocessor configuration where each ME executes all the tasks. Both these approaches have different pros and cons and typically a combination of the

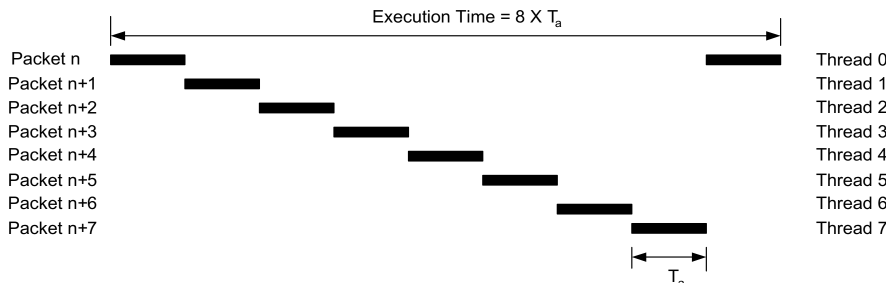


Fig. 2. Parallel processing of packets using multithreading within an ME.

two approaches is used to implement a fast path packet processing application.

### 4.2. Pipelining

In the pipelined approach, each function (or task) of a packet processing application is allocated to a different ME. As shown in Fig. 3, MEs are arranged in the form of a pipeline. Each packet traverses the pipeline of MEs.

As explained in the previous section, the total processing time for a function allocated to an ME is given by $8 \times T_a$. Each ME in the pipeline has the same amount of total processing time allocated to it. The total processing time for a pipeline of $n$ MEs is given by $8 \times n \times T_a$.

The advantages of using a pipelined approach are:

- The state for a given function that is persistent across packets (e.g., flow tables) can be held local to the ME in local memory or the local CAM. This eliminates the latency of accessing the state in external memory. This also eliminates the complexities associated with sharing the state with multiple MEs.
- The entire ME program memory space can be dedicated to a single function. This is important when a function supports many variations that can result in a large program memory footprint.

The disadvantages of using a pipelined approach are:

- The state that is "local" to a packet (e.g., updated packet headers) must be communicated from each ME in the pipeline to the next. This results in an undesirable amount of communication overhead if the packet state is large.
- Each function in the packet processing application must "fit" the worst-case performance requirements for that application. This may make partitioning the application into functions more difficult.

### 4.3. Multiprocessing

In the multiprocessing approach, each ME executes all the functions of the packet processing application. An array of MEs is used in parallel to simultaneously process packets as shown in Fig. 4. In this approach, each packet is only handled by one ME.

In a multiprocessing configuration with $n$ MEs, the total processing time for the sequence of functions executed by the ME is given by $8 \times n \times T_a$. Of this time, the total time for active execution of each thread is given by $n \times T_a$.

The advantages of using a multiprocessing approach are:

- Packet state can be held locally to an ME. This allows all the functions local access to the packet state and eliminates the latency of communicating this state between MEs.
- Each function need not fit the worst-case performance requirement for the application. The performance requirement on a given ME is for all the functions taken together. This allows some functions to take more time and use processing time from other functions that execute faster. This leads to a better overall utilization of the ME execution time.

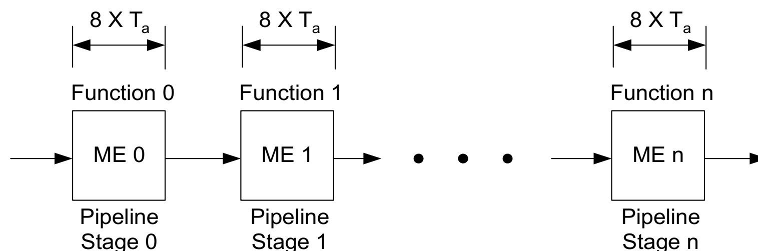


Fig. 3. Parallel processing using a pipeline of MEs.

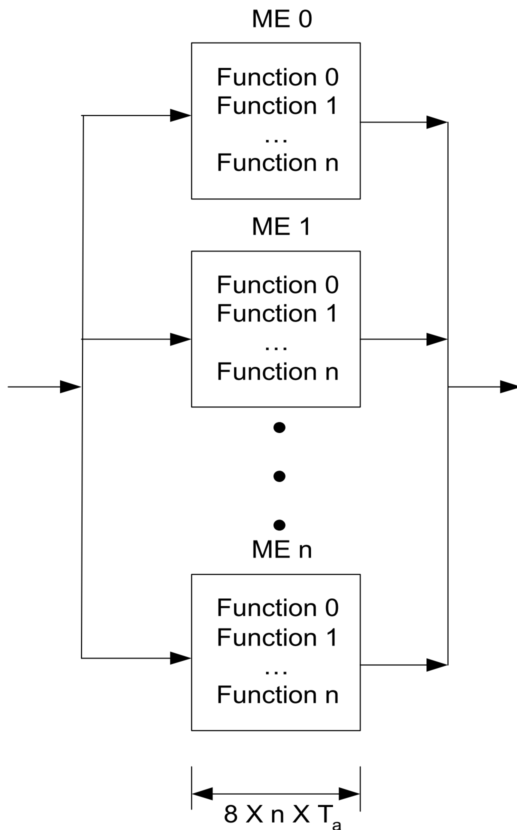


Fig. 4. Parallel processing using MEs in a multiprocessing configuration.

The disadvantages of using a multiprocessing approach are:

- The ME program memory space must be shared between multiple functions and can become a bottleneck when many functions are involved.
- Function state that is persistent across packets is kept in external memory. This makes accessing that state and maintaining the state coherency more costly.

### *4.4. Elasticity buffers*

A combination of pipelining and multiprocessing is typically used to implement a packet processing application. When a packet must be passed from one ME to the next, an elasticity buffer (implemented as a circular queue) is used. This elasticity buffer allows asynchronous execution between different parallel processing stages and accommodates jitter in packet processing. So if a processing stage falls behind in execution, due to a system anomaly such as an unusually high utilization of a memory unit over a short time period, the elasticity buffer will allow the packets to be processed by that stage to be buffered so that the previous stages are not stalled waiting for the next stage to complete. Statistically, the stage that falls behind will be able to catch up with its processing and system state will normalize.

The IXP2400 processor supports multiple methods for implementing elasticity buffers for communication between the pipe stages:

- SRAM/scratch rings: These are multiproducer, multiconsumer message passing queues maintained by hardware.
- Next neighbor rings: These are single producer–single consumer optimized message passing queues between adjacent MEs.

### *4.5. Critical sections*

An important consideration in the design of a packet processing application is the identification and implementation of critical sections. A critical section is a section of code where one ME thread has exclusive modification privileges for a global resource (such as a location in memory) at any one time. Different mechanisms are used to implement critical sections between the threads within the same ME and threads across different MEs.

*Critical sections between threads in a ME*

A critical section involves three steps:

- Reading shared data.
- Modifying the data.
- Writing back the modified data.

If the shared data is held local to an ME (e.g., in local memory), then the critical section is provided by the fact that threads' execution is non-preemptive. A thread executing a critical section can

swap out after it has completed the execution of that critical section thereby ensuring consistency of data.

If the shared data is held in external memory, a latency penalty will be incurred if each thread reads the data from external memory, modifies it, and writes the data back in sequence. This negates the benefits of multithreading because a thread cannot execute a critical section when a previous thread in the same critical section is waiting for an I/O operation to complete. To reduce this latency penalty associated with the read and write, the ME threads can use the CAM to fold multiple reads and updates to shared data into a sequence of "single read, multiple modifications and a single write" operations. The CAM is used to implement software controlled caching of shared data. Typically the first thread to enter the critical section fetches the shared data into local memory. Other threads use the CAM to determine that the shared data is in local memory and read and modify the data locally. The last thread evicts the data from local memory to external memory. The advantage of using the CAM is that the latency penalty for one read and write of the shared data is amortized over 8 (or more, depending on the cache hit rate) updates to that data. A second advantage of this approach is that this allows packet processing to be effectively performed in parallel even in the presence of high contention for shared state which can occur with a burst of packets belonging to the same flow.

*Critical sections between threads across different MEs*

To ensure exclusive modification privileges between threads across different MEs, the following requirements must be met:

- Only one ME must be executing a function with a critical section at one time. Within the ME that is executing the critical function, multiple threads can use the CAM as explained above to maintain coherency.
- The granularity of the function with the critical section must be such that this function fits the worst-case packet processing requirements for the application.

In the pipelined approach, each ME is assigned a different function and each function is sized to meet the performance requirements of the application. Therefore, mutual exclusion between MEs is not an issue.

In the multiprocessing approach, an ME should not begin executing a function with a critical section unless it can be ensured that its "previous" ME has transitioned out of the critical section. This can be accomplished by placing a *fence* around the critical section using inter-thread signaling. When all threads in a given ME are finished executing a critical section, the first thread of the next ME is signaled to begin executing the critical section.

In summary, several programming models are available to the application developer to develop the fast path packet processing applications on the IXP2400. A combination of the multithreading, pipelining and multiprocessing approaches can be used to implement the application on the array of MEs. One of several hardware-provided mechanisms can be used for elasticity buffers. Critical sections are realized using the ME CAM to implement software controlled caching. Packet order can be maintained using ordered thread execution or using packet sequence numbers. When a fast path application is mapped onto the IXP2400 microengines, the choice of the programming model should be guided by an objective to minimize the amount of state communicated between different parallel processing stages. This has the benefit that stages can run in parallel without too much synchronization between them and also the added benefit that the load on the system resources in the IXP2400 is minimized.

## 5. Example application: media gateway using TDM over AAL2

In this section we describe the design and implementation of a media gateway application on the IXP2400 and discuss some of the software challenges presented by this application. A media gateway is responsible for internetworking between the circuit-switched TDM voice network and the packet-switched network. The packet-

switched network can be either an ATM network or an IP network. The example application discussed in this section, assumes an ATM network. However, the design is equally applicable to a media gateway that interfaces with an IP network.

As shown in Fig. 5, an ATM media gateway connects the circuit-switched TDM network with the cell-switched ATM network. Several TDM voice circuits are aggregated onto an ATM virtual circuit using the AAL2 adaptation layer (called an AAL2 trunk). Several media gateways can be interconnected using a mesh of AAL2 trunks.

The packet processing flow in an ATM media gateway is shown in Fig. 6. The packet processing tasks are performed in two different domains. An array of media signal processors (MSPs—which are DSPs specialized for the task of processing voice [2]) handle the conversion of circuit-switched TDM into voice packets. The MSPs are responsible for the termination of TDM circuits, aggregation of TDM voice samples into voice packets at specific sampling intervals (e.g., 5, 10 ms, etc.), and voice signal processing for compression, echo cancellation, etc. The NP handles the aggregation of multiple voice channels into an ATM virtual circuit. As shown in Fig. 6, the NP is responsible for converting voice packets received from the MSPs into AAL2 CPS layer packets and multiplexing them onto an AAL2 trunk. In the reverse direction, the NP is responsible for creating voice packets, smoothing out the network induced jitter and playing back the voice packets to the MSPs at the specified sampling interval. The Fig. 6 shows the transformation of TDM voice into ATM cells. TDM voice channels are packetized into per-channel voice packets by the MSPs. The MSP adds a header to each voice packet to enable the NP to identify the voice channel and voice encoding

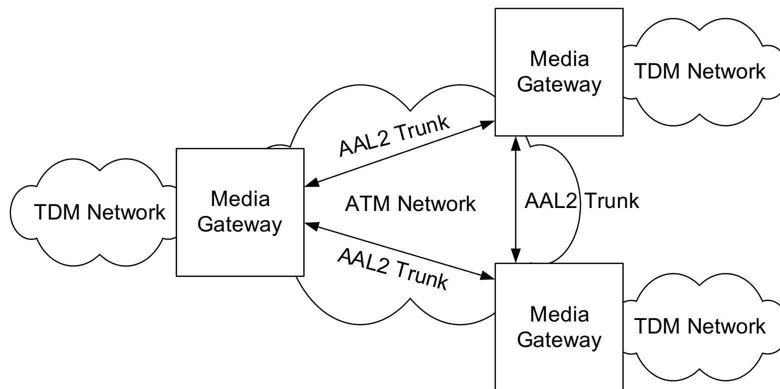


Fig. 5. Overview of a media gateway operation.

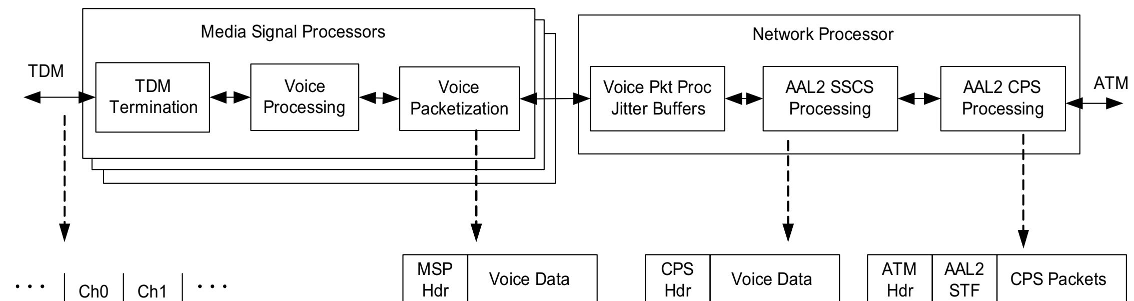


Fig. 6. Packet processing flow in an ATM media gateway.

information. Each voice packet is mapped into one AAL2 CPS packet by the NP. Multiple, CPS packets are multiplexed into one ATM cell.

In the next two sections, we focus on the NP and explain in detail how the ATM media gateway application was implemented on the IXP2400 network processor.

### 5.1. MSP to ATM processing design

Fig. 7 illustrates the major components, data structures and control and data flow for the MSP to 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.

The *MSP Rx* component receives voice packets from the MSPs. It reads the header that is created by the MSP and extracts the channel-specific information that describes the voice packet. The *SSCS Tx* component implements the ITU-T I366.2 AAL2 SSCS layer specification [5] for voice. 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 CID of this channel, the 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.

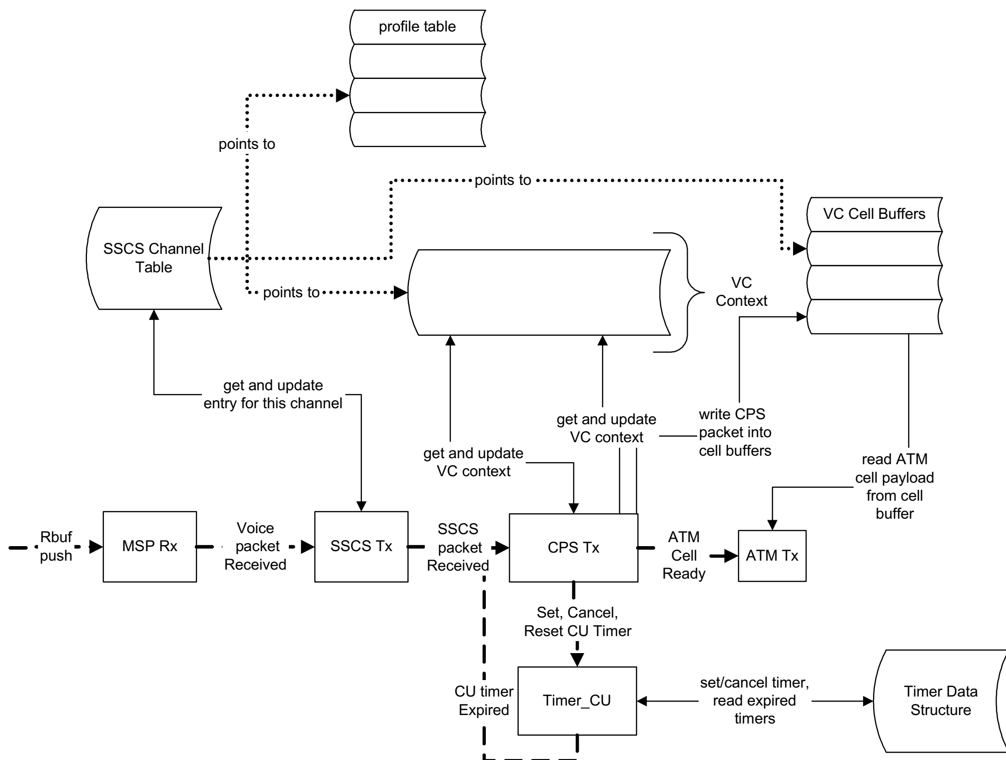


Fig. 7. MSP-to-ATM packet processing design.

The *CPS Tx* implements the ITU-T I363.2 specification [4] for the AAL2 CPS layer. It encapsulates SSCS packets in CPS packets and packs CPS packets into AAL2 cells. The packing of CPS packets into AAL2 cells is driven by the arrival of SSCS packets from the SSCS Tx component and also by the firing of the CU timer for the virtual circuit. The *Timer_CU* component implements the CU timer functionality. It accepts requests to set, cancel, and reset the timer for specific VCs. It is also responsible for firing the individual CU timers that are set (and canceled) for individual VCs. The *CU Timer Structure* is a calendar queue data structure used to store the CU timer entries for active VCs. The timers are stored in buckets, where each bucket is associated with a time interval. The Timer_CU component wakes up at the end of each time interval and sends CU timer expired messages to the CPS Tx component for each VC that had a CU timer set to go off during the previous time interval. Finally, the *ATM Tx* component performs the ATM header processing and transmits the cell.

### 5.2. ATM to MSP design

Fig. 8 illustrates the major components, data structures, and control and data flow in the ATM to MSP direction. The *ATM and CPS Rx* component receives ATM cells. The ATM and CPS Rx component reads the contents of the ATM cell into the ME and steps through the CPS packets contained within. It copies each CPS packet into a packet buffer, maps the VPI/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.

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. This profile table entry specifies the encoding algorithm that was used, and 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 time stamp. If there are no errors, the packet is passed to the SSCS2MSP component. The *SSCS2MSP* component creates the MSP packet header from the information that the ATM and CPS Rx and SSCS Rx components have extracted from the packet

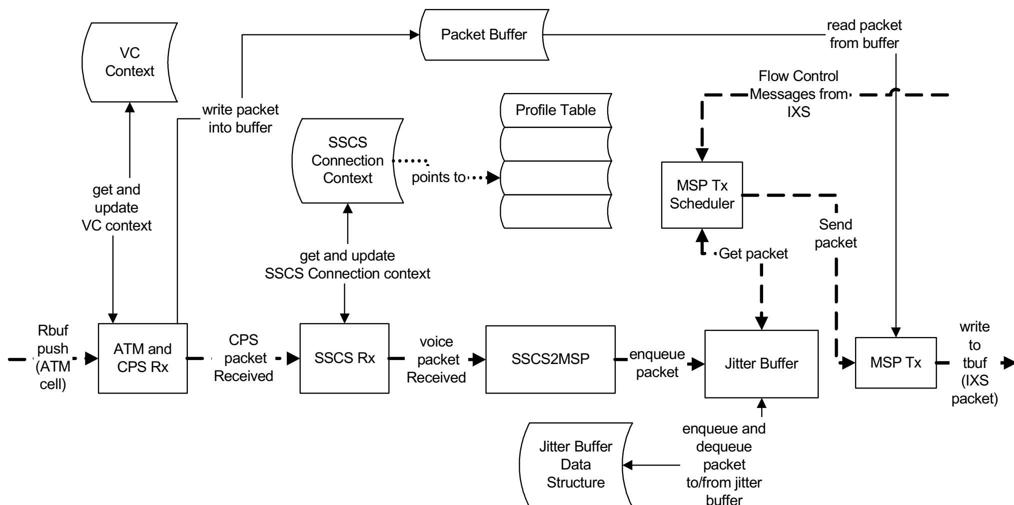


Fig. 8. ATM-to-MSP packet processing design.

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 *MSP Tx Scheduler* component is responsible for scheduling the transmission of MSP packets to the farm of MSPs. The transmission of MSP packets is triggered by periodic requests received from the farm of MSPs for voice packets from a specific channel. For each requested channel, the MSP Tx Scheduler component asks the jitter buffer component to dequeue a packet from the jitter buffer. Finally the *MSP Tx* component transmits voice packets to the MSP farm.

### 5.3. Challenges

In this section we discuss some of the challenges that must be surmounted in developing a media gateway application on a NP.

#### 5.3.1. Processing asynchronous inputs within many contexts

The ATM media gateway 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 MSP to ATM direction, voice packets destined for a VC should be processed in order, while in the ATM to MSP 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 CU timer. CU timer expiration events are neither regular nor 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 CU timer.

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 ATM media gateway 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 distribute 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.

#### 5.3.2. Bit- and byte-level memory access

The CPS header and AAL2 cell are tightly packed structures, where fields are not aligned on 4-, 8-, or even 1-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 4- or 8-byte chunks of data, addressed on 4- or 8-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 masks 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.

#### 5.3.3. 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 coded 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 microengines. 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 microengines. 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.

## 6. Example application: ATM traffic management

### 6.1. Application overview

Traffic management in ATM networks is specified by the ATM forum in the Traffic Management Specification [9]. The TM4.1 specification defines six service categories, which 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 to be provided to the traffic in the category, provided the traffic conforms to the specified traffic contract.

The key ATM service categories are as follows:

1. The constant bit rate (CBR) service category provides a service similar to that provided by a TDM circuit. CBR traffic is characterized by a peak cell rate and cell delay variation tolerance. A conformant CBR traffic stream cannot exceed its peak rate or a maximum cell delay variation tolerance. The traffic is guaranteed very low losses and a maximum cell transfer delay. Voice and circuit emulation services are potential users of this service.
2. 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 is sustainable rate over long timescales and 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.
3. The nonreal-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.
4. The unspecified bit rate (UBR), available bit rate (ABR) and guaranteed frame rate (GFR) service categories are intended for transport of data traffic. Simple-UBR is a best effort service, where no restrictions are placed on the traffic and the network provides no guarantees. There are other variations of UBR like UBR with PCR shaping, UBR with Min desired cell rate shaping and differentiated UBR that do provide some minimal guarantees for the traffic. 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 (UPC) is performed at the edges of the network to make certain that the traffic entering the network conforms to its traffic contract. Finally, shaping is used to transform a traffic stream into one that meets a different traffic contract.

The concept of policing and shaping are illustrated in the context of a DSL Access Multiplexer (DSLAM) in Fig. 9. We have several customer premises equipment (CPE) which connect to the DSLAM on their unique virtual circuits (VC-1 to VC-$n$). The number $n$ can be very large depending upon the number of CPEs. In the simplest case, each of the CPEs will have a pre-negotiated, end-to-end traffic contract for their VCs along with the service category. Often, the CPEs do not adhere to this traffic contract and burst cells at rates faster than they are supposed to. The DSLAM imple-

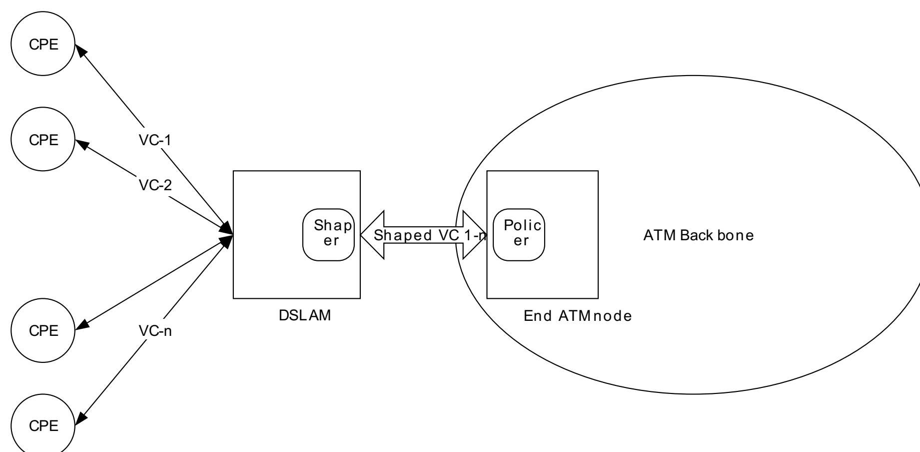


Fig. 9. Concept of ATM traffic management in the context of a DSLAM.

ments traffic shaping on the egress path, where it needs to make sure that the traffic on the individual VCs going out the wire toward the ATM backbone conform to the traffic contract.

The end ATM node in the ATM backbone monitors the cell arrival patterns on the individual VCs. If the traffic shaping by the DSLAM does not conform to the pre-negotiated traffic contract, the policer at the ingress of the end ATM node drops the cells that do not conform to the traffic profile.

### 6.2. Implementation overview

Several ASIC-based traffic management solutions exist today [1]. In this section, we describe an implementation of traffic shaping on the IXP for the basic service categories of CBR, VBR and UBR. The basic algorithm used for shaping is the generic cell rate algorithm (GCRA). GCRA has two parameters $T$ and $\tau$. The inverse of the first parameter represents the rate allocated to the flow by the network. The rate here could mean either peak rate or average rate depending upon the service category. The second parameter $\tau$ represents the tolerance around the theoretical arrival times of the cells in a flow. The goal of GCRA is to make sure that the bandwidth used for the traffic from the given flow never exceeds the pre-configured rate and the traffic never violates the tolerance limits in the cell arrival times. The end result of shaping is the "earliest departure time" for the cell being shaped.

The actual shaping algorithm, which we implement in the ME software is as follows:

```
GCRA(T, τ) on the arrival of a new cell:
  Working variables:
    t = actual cell arrival time
    TAT = theoretical cell arrival time
    t1 = earliest departure time
  If (t > TAT − τ)
    t1 = t;
  Else /*(t < TAT − τ)*/
    t1 = TAT;
GCRA (T, τ) on the departure of a cell:
  Working variables:
    td = actual cell departure time
    TAT = theoretical cell arrival time
  If (td > TAT)
    TAT = td + T;
  Else /*(td <= TAT)*/
    TAT = TAT + T;
```

Once the earliest departure time is obtained for a cell, we need to store these times in an efficient manner and schedule the cells based on these

times. The key concept for the scalable implementation of such a data structure is the concept of time queue, which we implement in software.

### 6.3. Time queues

*Motivation*

Traffic shaping based on TM4.1 is essentially a non-work conserving scheduling policy. The cells from the contract-based VCs like CBR-VCs and VBR-VCs should not be scheduled simply because there are cells available on the VCs and the outgoing link bandwidth is idle. Instead, the cells need to wait in the transmission queues until the departure time for these cells has arrived.

Each cell takes a fixed amount of time to go out on the wire. We call this time, a cell transmission slot. The infinitely long time axis can be viewed as a sequence of endless cell transmission slots as illustrated in Fig. 10. We can however only store a certain number of these endless cell transmission slots in memory. This finite number that we store in memory at any given point in time is called the *time horizon*. Once a cell transmission slot expires in the current time horizon, it becomes the latest cell transmission slot for the next time horizon. As shown in Fig. 10, time slot $a$ in the current time horizon, upon expiry transforms to timeslot $a'$ for the next time horizon.

In order to successfully implement conformant traffic shaping functionality, we need to convert the above concept of time horizon into the time queue data structure.

*What is a time queue?*

Time queues are implemented in software and are depicted in Fig. 11. Each time queue is nothing but an aggregation of a programmable number of cell transmission slots and hence represents an interval in time. The aggregation is performed to reduce the total number of time queues and to efficiently handle the resulting data structure complexity.

Each time queue holds all the cells that are intended to depart the system during this time interval. Thus, a set of time queues represents a long time interval. As time passes, one time queue is emptied of cells that get transmitted and the clock advances to the next time queue. Eventually, the clock will wrap around and the time queues will get reused with each time queue representing a new time interval in future. Some of the time queues may have more cells queued up than what could be transmitted in the time interval, since there is no admission control mechanism devised for queuing cells into the time queues. Further, each port has two sets of time queues, in order to differentiate the cells in terms of delay and loss properties of the service categories. The scheduling of cells from these two sets of time queues is in strict priority order. The resulting data structure is illustrated in Fig. 11.

The concept of aggregating the cell transmission slots into a time queue has its own advantages and disadvantages. The main advantages are that the number of time queues is reduced and one can also expect to see a good averaging behavior in terms of the occupancies of these time queues. The disadvantage to aggregating cell slots into time queues is that there would be non-determinism in the departure times of the cells. A cell that needs to depart later than time $t$, could depart as early as $(t - N + 1)$, where $N$ stands for the aggregation factor of the time queue. If $N$ is chosen properly,

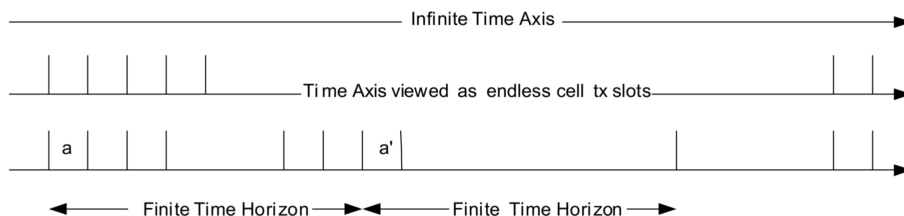


Fig. 10. Concept of the time horizon.

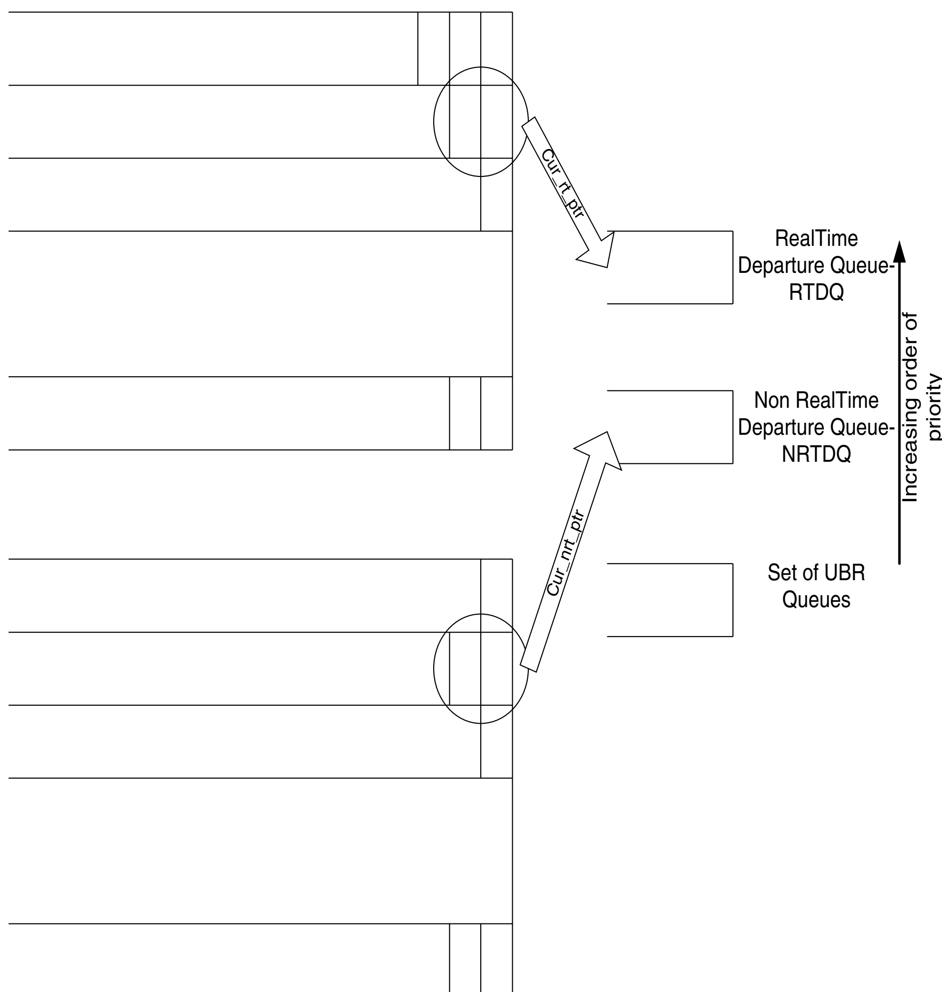


Fig. 11. Time queue data structure.

this amount of non-determinism in the departure times can be kept within an acceptable limit. It is also to be noted that setting $N = 1$ would totally eliminate this problem.

With the view of pros and cons of this approach in mind, our implementation of the TM4.1 standard proposes that $N$ be made programmable with some default value.

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 is nothing but the time horizon. Here is an example: If the slowest VC rate supported in the system is $\mathrm{VC_{slow}}$, the aggregation level of each time queue is $N$ and the link rate is $LR$, then the number of time queues is given by $2 \times \mathrm{LR}/(\mathrm{VC_{slow}} \times N)$. The factor of 2 is needed because we have two sets of time queues, one for real time traffic and the other for non-real time traffic. The real time traffic gets strict priority over the non-real time traffic.

Given that each time queue has an aggregation of $N$ cell transmission slots, the architecture does not provide a means to guarantee that no more than $N$ cells will be stored in that time queue.

Instead each the time queue is implemented as a linked list in software to accommodate the cases where we could have more than $N$ cells stored in a time queue. This approach simplifies the design a lot, avoiding unnecessary timers and state maintenance, and makes it highly scalable to a large number of time queues (i.e., slower VC rates than the slowest one assumed).

### 6.4. Software architecture for traffic management

Fig. 12 shows the overview of the architecture. We have divided our traffic management solution in four modular blocks, namely, queue manager (QM), shaper, writeout and scheduler.

The ATM cells flow through this pipeline as follows (Fig. 12):

- The buffer management (BM—not part of the TM4.1 blocks) module is responsible for deciding whether to accept a cell or a frame (in case of AAL5) based on a mechanism such as Weighted RED [8] or simple tail-drop. For simplicity, we will refer to each unit of data as a packet (could be a single cell or a frame reassembled out of multiple cells), If the packet is accepted, the BM module passes it onto the QM for queuing in the appropriate queue based on the packet's virtual circuit identifier.
- The QM is responsible for both enqueuing the packet and dequeuing the cells of the packet. The QM maintains a set of queues called the virtual circuit queues (VCQ). Each ATM virtual circuit will have a corresponding VCQ. The QM lets the shaper know of all the enqueues and dequeues in a transparent manner.
- The shaper operates in a slightly different manner for the enqueues and dequeues that happen via the QM. When there is a fresh enqueue into a VCQ that has no data, the shaper applies GCRA based shaping algorithm for the VCQ, to obtain the earliest departure time of the head of the line cell from the VCQ. It then communicates this time to the writeout block. When there is a dequeue from the VCQ, the shaper first updates the cell theoretical arrival time, based on

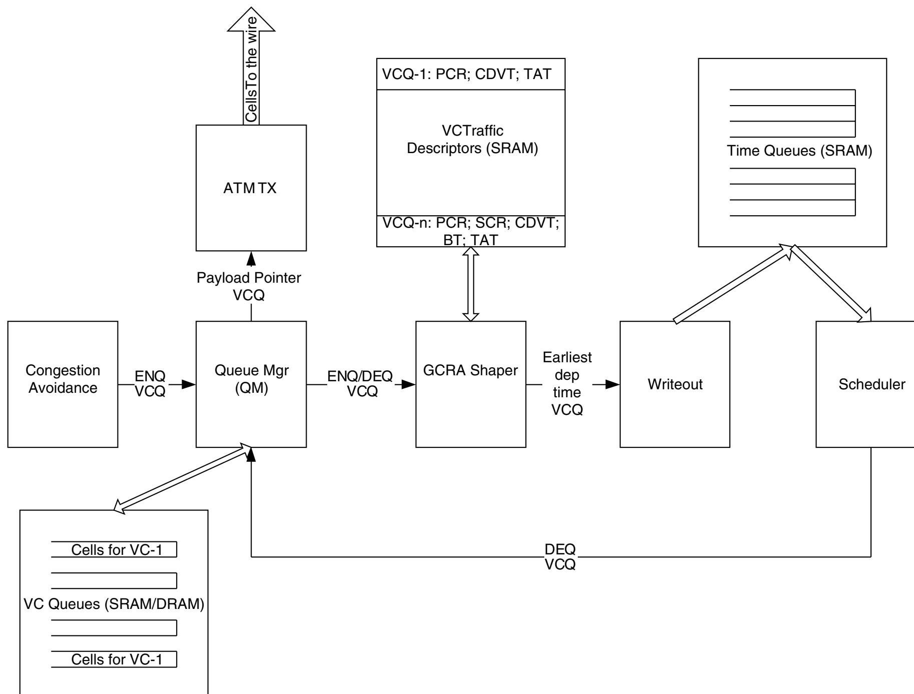


Fig. 12. Software architecture overview for ATM traffic management.

the departure time of the last cell. If there are any cells present in the VCQ after the dequeue happens, the shaper executes the GCRA shaping algorithm and obtains the earliest departure time of the head of the line cell and communicates the time to the writeout block.

- The writeout block receives the earliest departure times of head of the line cells in the various VCQs, from the shaper. It then computes the time queue into which the cell has to be written based on the earliest departure time and the service category. If the traffic is rt-VBR or CBR, the VCQ to which the cell belongs is written into the highest priority time queue (called real time time-queue). If the traffic is nrt-VBR, the VCQ to which the cell belongs is written into the second-priority time queue (called non-real time time-queue). Once the write is completed, the length of the corresponding time queue is incremented. The way time queues are designed, there could be many VCQ entries that get written into the same time queue. We do not use complicated timers to track the occupancies of the various time queues.
- Shaper and writeout operations are invoked for all traffic classes, except simple-UBR.
- The scheduler simply traverses the time queues and schedules cells for transmission by reading the VCQ entries from the two sets of time queues per port, based on the current time. If there are no entries available for scheduling on that port for that time-slot, the scheduler then checks the simple-UBR queues for that port. There are a configurable number of simple-UBR queues per port and the scheduler schedules from them in priority order. Whenever there is flow control asserted on a particular port, no cell is transmitted from that port.
- Once the scheduler determines the VCQ from which the cell needs to be scheduled, it informs the QM of this VCQ. The QM then goes ahead and dequeues the pointer to the buffer descriptor for the head of the line cell for this VCQ. It then communicates this pointer to the transmit block.
- The transmit block does the actual transmission of the cell payload onto the wire through the media switch fabric.

*Problems with very high bit rate VCs*

The basic algorithm described above runs into some problems for high bit rate VCs, the configured rates for which are comparable to the line rate. The reason for this is as follows:

The head of the line cell of each VC is written into the time queue. Once this cell is scheduled out from the time queue, the next cell from this VC is shaped and written into the time queue and so on. Since the time queues are stored in SRAM, there is a certain amount of latency involved in writing and reading from the time queues and providing feedback for the next cell to be shaped on this VC. If this feedback latency happens to be greater than the inter-departure times of the cells of the VC (which is very small for high bit rate VCs), we will not be able to guarantee the QoS for this VC. Hence the SRAM time queues have a problem for very high bit rate VCs, whose cell inter-departure times are comparable to the inherent read-write latencies of the NP.

*Solution for high bit rate VCs*

The solution for high bit rate VCs is to use the local memory extensively, knowing that the turnaround times of such VCs are very small and hence can be fit into local memory.

The idea is to maintain a separate time queue for high bit rate VCs in local memory. This time queue is a static schedule of high bit rate VC transmission slots that wrap around in a circular fashion. The number of entries in this time queue is governed by the rate of the "slowest" high bit rate VC. Each slot of this time queue is pre-computed with a VC entry. The length of the high bit rate VCs is also stored in local memory. At the beginning of every cell transmission slot, the high bit rate time queue is examined to see which high bit rate VC is eligible to be transmitted. If there are no cells to be transmitted from this queue, cells are transmitted from the SRAM time queues or the UBR queues.

### 6.5. Details of the software blocks

*Queue manager*

The QM block maintains all the VCQs in the system. The VCQs are stored in SRAM. The

payload enqueued in the VCQ is actually stored in DRAM. The QM enqueues and dequeues the pointer to this payload into the VCQs in SRAM. The SRAM offers hardware support for efficient enqueing and dequeing payload pointers from the VCQs by providing a 64-entry queue array cache.

The number of VCQs can be very large, and hence the QM operates by caching the state of the 16 most recently used queues in the hardware queue array of the SRAM controller. It can then enqueue and dequeue into the cached queues using single instruction commands. There are no inherent scalability limits on the number of queues than can be stored in the QM due to this approach.

The main function of the QM is to enqueue frames or cells into the VCQs and to dequeue cells from the VCQs. Whenever a new frame or a cell needs to be enqueued, the pointer to this payload is provided to the QM along with the VCQ into which the cell has to be enqueued. The queue manager checks its local CAM to see if the VCQ is cached as a hardware queue. If not, it evicts the LRU queue from the hardware cache and brings in the required VCQ. Once the required VCQ is available in the hardware cache, the QM enqueues the frame or the cell using a single instruction enqueue command.

The dequeue operation happens on the same lines. The key point to note however is: the QM can effectively virtually chunk up a frame into constituent cells (needed for AAL5) by the concept of "cell dequeue", without spending computes on physically chunking up the frame. In order to achieve this, the QM maintains a count of the number of cells in the enqueued frame. It then dequeues the pointer to this frame until this count reaches zero.

The QM also communicates all the enqeue/dequeue information to the shaper along with the queue state transition information—i.e., whether the queue cell count went empty after a dequeue operation on the VCQ or whether it became non empty after an enqueue into the VCQ.

*Shaper block*

The TM4.1 shaper receives input from the QM upon enqueue and dequeue. In addition, it receives the number of cells enqueued and also information on whether there has been a queue state transition for this queue.

By looking at the VC number, the shaper gets to know whether the VC is high bit rate or low bit rate. For low bit rate VCs, the shaper determines if there has been a transition of the queue state (from empty to non-empty or vice versa) from the message. If there has been a transition, and the message was a enqueue message or if there has been no transition and the message was dequeue message, the shaper invokes the GCRA shaping function.

The shaping function computes the *earliest* departure time for the cell using the GCRA algorithm. It passes on the *earliest* departure time and the VC number to the TM4.1 writeout block.

For high bit rate VCs, the length of the VCQ and the VCQ are communicated to the writeout block.

*Writeout block*

For the low bit rate VCs, the write out block computes the time queue into which the cell needs to be written based on the *earliest* departure time that it receives from the shaper. 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.

For high bit rate VCs the, writeout updates the cell count of this VC queue in the local memory.

Write-out and scheduler blocks exist on the same ME so that they can share common state like the lengths of the high bit rate VCs.

*Scheduler block*

The scheduler maintains two data structures—the real time departure queue (RTDQ) and the non-real time departure queue (NRTDQ). These two data structures are maintained as software rings in the local memory. As real time ticks and cells are transmitted, every $N$ cell transmission slots, the cells of the current real-time time queue in SRAM move to the RTDQ and the cells of the current non-real time time-queue move to the NRTDQ. RTDQ has a higher priority over NRTDQ. Note that cells do not have to be physically moved from the TQ to the DQ—moving the TQ pointer into the DQ would be sufficient.

The scheduler block schedules out cells from the departure queues and the simple-UBR queues. It is

essentially a priority mechanism with the highest priority for the high bit rate time queue, followed by RTDQ, NRTDQ and the simple-UBR queues. Other alternatives that could be implemented for scheduling among the simple-UBR queues are weighted fair queuing, round robin scheduling, etc.

### 6.6. Key challenges and conclusions

In the previous few sub-sections, we described the implementation of an ATM TM4.1 based traffic shaping algorithm. The following is the summary of the key challenges that need to be addressed for such an implementation:

1. *Scalability with the number of VCs:* In a realistic ATM traffic management system, one could have several thousand VCs. Care should be taken to ensure that all the components of the traffic management system are scalable with the number of VCs. We address this challenge by making the queue manager inherently scalable to any number of VCs. Also, the time queue data structure is not dependent of the number of VCs, but just the slowest VC in the system.
2. *Scalability with the VC rates:* The solution needs to be scalable to a single VC running at the output wire rate and at the same time down to several VCs running at a preconfigured minimum VC rates. We address this challenge by making sure that the time horizon of the system covers the cell inter-departure times of the slowest VCs. For scalability to high bit rate VCs, we have a solution that bypasses the feedback loop latencies between the shaper and the scheduler blocks.
3. *Cell order maintenance:* We need to ensure that the traffic management system does not re-order cells on each of the VCs. Cell re-ordering can have disastrous effect on the application performance and throughput at the end hosts. This challenge is addressed by following the hyper task chaining programming model (Section 4).
4. *Achieving required performance:* The traffic management system needs to guarantee the targeted performance. In other words, the traffic management software needs to be able to handle cells arriving back to back at the wire rate. In such a scenario, we need to make sure that each of the ME finishes its processing every cell arrival time and hands off the cell to the next ME. The cell arrival time translates to a fixed number of processing cycle budget per ME. We make sure that the layout of the software blocks on the MEs is determined by the processing cycles requirement for each of the blocks and the sum total of the cycle requirements of the blocks mapped onto an ME is less than the processing budget available on the ME.

## 7. Conclusions

NPs are an emerging field of programmable processors that are optimized to implement packet processing functions in networking devices. The programmable nature of NPs also brings its own challenges. To handle multiple media interfaces and/or to handle data units arriving back-to-back at high data rates, an NP must perform fast I/O and memory operations such as packet storage, table lookup, and extracting fields in packet headers. This paper provides an overview of Intel's second-generation NPs, the programming model involved, and describes how to use the model to implement two applications on a distributed multiprocessor architecture. The programming model and examples of applications demonstrate the power and flexibility of the distributed, multiprocessor architecture of the IXP 2000 series family.

## References


[1] H.J. Chao, J.S. Hong, Design of an ATM shaping multiplexer with guaranteed output burstiness, Int. J. Comput. Syst. Sci. & Eng., Special issue on ATM Switching, 12 (2) (1997) 131–141.

[2] Intel Medial Signal Processor: <http://developer.intel.com/design/network/products/wan/vop/IXS1000.htm>.

[3] Intel® Network Processors: <http://developer.intel.com/design/network/products/npfamily/index.htm>.

[4] ITU-T Recommendation I.363.2, Series I: B-ISDN ATM Adaptation Layer Specification: Type 2 AAL, ITU, Geneva, Switzerland, 1997.
[5] ITU-T Recommendation I.366.2, AAL Type 2 Service Specific Convergence Sub layer for Trunking ITU, Geneva, Switzerland, 1999.
[6] P. Chown, Advanced encryption standard (AES) ciphersuites for transport layer security (TLS), RFC 3268.
[7] P. Karn et al., The ESP triple DES transform, RFC 1581.
[8] S. Floyd, V. Jacobsen, Random early detection gateways for congestion avoidance, IEEE/ACM Trans. Networking 1 (4) (1993) 397–413.
[9] Traffic Management Specification, Version 4.1. ITU Document#: AF-TM-0121.00.

**Muthu Venkatachalam** is a network architect with the communications group at Intel Corporation in Portland, Oregon. His interests lie in network processor architectures, QoS algorithms, traffic management, systems modeling and analysis, and keeping pace with the innovations in today's networking industry. He has worked on traffic management solutions, processing architectures for Intel's next generation Network processors, architectures for broadband and wireless access systems and metropolitan optical networking systems. He has a bachelor's degree from the Indian Institute of Technology at Kharagpur and a graduate degree from the University of Texas at Austin. He can be reached at muthaiah.venkatachalam@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

**Raj Yavatkar** is currently the Chief Software Architect for the Internet Exchange Architecture at the Intel Communications Group. Previously, Raj led the advanced R&D activities in the areas of programmable networks, policy-based network management, and end2end Quality of Service (QoS). He initiated and led the definition of a new industry-wide framework for policy-based networking that resulted in both the development of IETF standards (e.g., COPS) and the technology that Intel licensed to HP OpenView and others.

Raj Yavatkar received his Ph.D. in Computer Science from Purdue University in 1989. He has numerous publications to his credit and serves on the technical program committees of leading conferences and workshops. He also co-authored the book "Inside the Internet's Resource reservation Protocol (RSVP)" published by John Wiley in 1998. He currently serves on the editorial board of the IEEE Network magazine.