Open Programmable Layer-3 Networking

Hardware Approach for Full Active Network

Koji HINO, Takashi EGAWA, and Yoshiaki KIRIHA
NEC Corporation

Key words: Active Network, Programmable Layer-3, High speed Active Layer-3 Processor

Abstract: This paper discusses high speed (multi-gigabits/second) active networking techniques used to introduce active functionality into layer-3 processing in order to provide wide-ranging flexibility to internetworking. We propose here StreamCode, a compact object code for layer-3 programming, a StreamCode Processor that achieves a high-speed execution of the object code, and an architecture of StreamCode-based high-speed active networking node. The processor has a unique instruction fetch mechanism to prevail over promiscuous instructions and data flows inside the processor, and has resource management function to execute StreamCode programs safely. Our FPGA based prototype system with sample applications confirmed the feasibility of proposed StreamCode based programmable Layer-3 networking.

1. INTRODUCTION

An active network technology is aimed to bring remarkable flexibility and extendibility to future networks, and many discussions from the several technical viewpoints such as language, node OS, security, and performance questions, have been made [1-5]. Some studies suggest controlling layer-3 functions (i.e. routing table searching, packet forwarding, and queueing management, etc.) in an active networking framework. The authors also believe that layer-3 activeness must be seriously considered. This is because most of current and proposed network services (ex. QoS control[6], VPN, IPv6[7], DiffServe[8]) require some modification to layer-3 functionality on many network nodes.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35522-1_37

H. R. van As (ed.), *Telecommunication Network Intelligence*

Yet, however, research on active layer-3 control, especially for a high speed (multi-gigabits/sec) networking environments has been limited. To make future layer-3 networks active, and to utilize the activated layer-3 networks into practical as current IP networks, we have to introduce new layer-3 networking systems whose performance (packet/sec, bandwidth for actual application contents) is comparable to non-active networking systems, with enough programmability, security, and stability.

We propose here a hardware-based layer-3 active networking system, aiming to achieve multi-gigabits per second link throughput. It consists of two sections: a packet-by-packet active code execution environment (an active layer-3 section), and an ordinal active network execution environment (an active node section). We are focusing on the active layer-3 section, and interaction mechanisms for these two sections in this paper.

For the active layer-3 section, we have designed an object code, 'StreamCode,' and a processor for it. StreamCode is a machine language level binary object code with which users can program layer-3 actions according to packet sender's idea, and the object code is executed by dedicated StreamCode Processor. StreamCode fragments are executed shortly, only while that packet (which contains the StreamCode fragment) is stayed at an input buffer of StreamCode Processor. So functionality and complexity of each StreamCode program is essentially limited. To compensate for this limitation, we have introduced interaction mechanisms between StreamCode execution and active node section (on network nodes, servers, and terminals).

Section 2 of this paper, we discuss related research, in Section 3 we introduce the basic concept of an active layer-3 networking architecture, and in Section 4 we describe our StreamCode based networking architecture in detail. In Section 5, we introduce the StreamCode Processor, which is dedicated to activating layer-3 functions, in Section 6, we briefly describe our hardware and software based prototype system, and in Section 7, we summarize our work.

2. RELATED WORK

There are basically two approaches to active networks: the programmable switch approach, and the capsule approach [3]. In the programmable switch approach, layer-3 functionality is classified as a target of control for active programs or open programmable APIs, and actual switches are ATM or IP switches with externally controllable interfaces. Current research efforts are focused on controlling existing layer-3 parameters. With respect to the capsule approach, researches are currently

focused on program execution itself, however, the next step will have to deal with autonomous layer-3 functionality, and capsule execution environment will require layer-3 activeness.

This paper introduces StreamCode, which is very influenced by capsule approaches. StreamCode has roughly the same packet format as does the capsule approach, with programs located in each packet, and nodes have execution units for these program pieces. There is a difference, however, in their respective execution environments. Capsule runs on top of general purpose unified operating environment, while StreamCode runs inside an active layer-3 execution environment.

Four studies related to active layer-3s are particularly important with respect to our work. First, [9] proposes to download transport active modules into network nodes. Although this basic concept is the same as ours, the approach is different in the sense that active modules are defined as node-vendor-supplied proprietary code modules, which offer standard API to modify its transport behavior. In our approach, the sender of code implementing active L3 module is defined as end user's applications, including servers and clients. Second, [10] describes importance of active-execution's performance for practical active network. The results verify that using processor's native object code will boost three times faster than Java bytecode. This fact shows that indirect execution (i.e. interpreting portable object code by general purpose processor) of active programs is slower than hardware based native execution, and we believe the fact proves the appropriateness of our hardware based active networking approach. Third, [14] presents Field Programmable Gate Array (FPGA) based protocol processor. This idea will provide programmable hardware inside layer-3 function with reasonable performance: node vendor and/or network provider can change hardware function, and end user can request to select hardware functions. However, FPGA has a limitation in its reconfiguration delay, thus packet-by-packet level activeness is hardly achieved with this approach. In our approach, each packet has different active program to realize application-driven network progression (uncountable kind of active programs), in expectation of high-speed wired-logic processors. Nevertheless, we are very interested in the study, because our design will eventually need programmable co-processor framework. Fourth, [11] introduces high performance active node targeting multi gigabits/sec level link speed.

Especially, fourth one's target environment is very close to ours, except that they assume that 'flow' can be managed by each node and use these flow information to bypass active processor. Their architecture may be resulted from their precondition, active packet to non-active packet ratio is not so high, and / or each flow shares its active algorithm. However, we

expect that active network aware application should use a different kind of active programs even for single flow, according to their need. Furthermore, we assume that there will be too many (ex. billions) flows aggregated on core routers, to be distinguished in a timely manner. Taking into account such assumptions, our approach chooses to abandon flow detection and all packets must contain program to declare its behavior. Each router's layer-3 function is 'passive' execution engine, and has no autonomous functions for flow detection, no search functions for packet handling program/data, no application-specific programmed behavior, etc.: all actions of the router are described in each incoming packets.

3. HOW TO ACHIEVE PROGRAMMABLE LAYER-3

To make layer-3 functions active network friendly, we have to develop interfaces from active modules to layer-3 functions, as shown in Figure 1. In the figure, stars are some part of layer-3 functions, which serve active network aware actions. Here, we call these as active L3 modules. Active L3 modules must be under control of a node OS, and their control interface will be provided by the node OS, as these functionalities are thought as important network resources. There are three possible ways to implement active L3 modules.

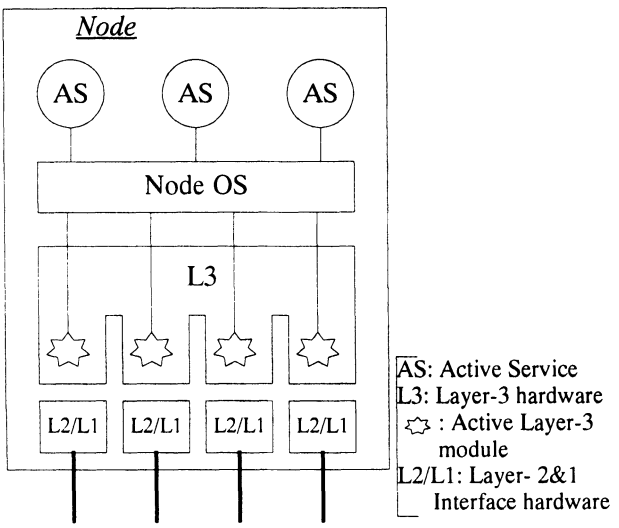

Figure 1: Active Layer-3 module

a) Stop improving layer-3: A functionality and interface of an active L3 module are specified and fixed forever. Any active services can change layer-3 behaviors through the interface of node OS. Active L3 module itself is developed and installed by the node vendor. This plan is the most practical and rational answer, and upgrades current network to active network, but we hope no future functional improvement of layer-3 except for its speed and cost. Ironically, as active network spread widely (we hope so), functional improvement of layer-3 will be discouraged, and would not be allowed. As a result, active service cannot use these improvements.
b) Drop-in layer-3 module: Although an active L3 module's functionality is not specified, standard interfaces for installing new active L3 modules, controlling these modules, and uninstalling these modules will be introduced. Functions of each active L3 module will, as same as network service build upon IP, consist of standard functions, private functions, and to-be-standard functions. However, since active L3 modules are depending on each node's architecture (i.e. proprietary), they are not reusable over the whole network. This is a good answer to balance flexibility and performance, because node vendor can tune these active L3 modules to their products. This plan will be implemented by network processors like TOPcore[13]. On the other side, if a node vendor would not want to make active L3 modules, or another node vendor would discontinue their business, no more active L3 module will be developed.
c) Make active layer-3 execution environment: A new active networking environment for active L3 modules will be introduced. It should be deployed not on the top of the node OS, but inside node's layer-3 functions. Active L3 modules are written in standard languages, and the execution environment must provide abstract and standard functions for active L3 modules to describe specialized layer-3 functions. With the environment, although we can achieve maximum flexibility, standardization effort on a language and API for active L3 modules should be necessary. Active L3 modules can be downloaded through node OS, or can be transferred as a program piece in each packet like capsule-type active network. An active L3 module, as a part of layer-3 functions, will be selected and executed upon arrival of each packet, so severe requirements for module selection delay and module execution throughput are exist. These requirements disallow us to integrate active L3 module execution environment to ordinal active node OS environment in a unified fashion. Because of this, two execution environments will have to coexist and interact with each other.

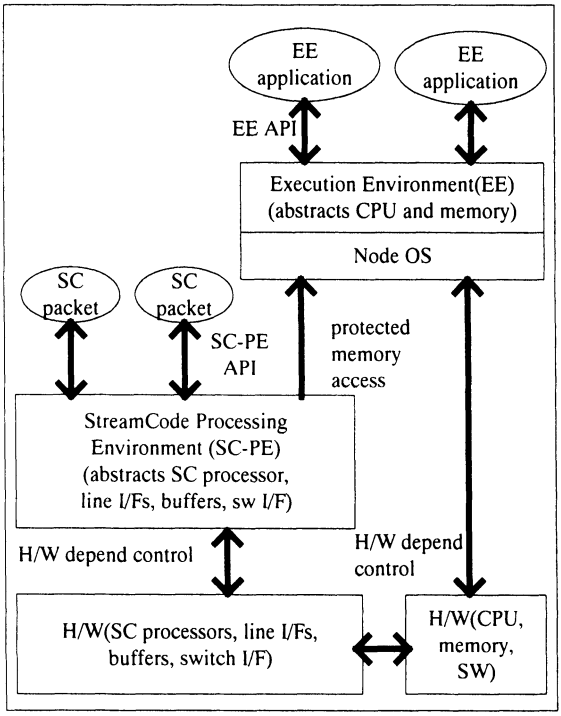

Figure 2: Functional Architecture of StreamCode Node

Taking into account the above each consideration, we have chosen answer c, because of its maximum flexibility and challenging goal. In order to shorten active L3 module selection delay to appropriate level such as sub-microsecond order, we have to decide how the implemented code of active layer-3 module will be delivered to each router. There are three options: 1) manual downloading by applications, 2) automatic fetching from code servers, 3) attaching code to every packet. Option 1 has a scalability problem because manual downloading implies that node has to manage code memory in a hard state manner. Option 2 introduces inadmissible delay before starting execution, if there were the unlimited number of code modules. Option 3 has a problem on code size overhead, but has no delay to start execution. We decided to take option 3 as a result that we regard the code execution speed as principal, although we should persevere with traffic inefficiency caused by each code fragment. We note that decreasing transmission cost and longer MTU of layer-3 packet will help to ease this disadvantage, and to make fruitful active network, millions kind of code fragments should be used over the network.

4. STREAMCODE BASED ACTIVE NODE ARCHITECTURE

Based on the previous discussion, we propose an active L3 system, named 'StreamCode' based active node. A StreamCode program implements an active L3 module, which is included in every packet and is executed by dedicated hardware (StreamCode Processor) in a non-blocking fashion. Due to this processing principle, service specific calculation only implemented by long-lived algorithms (ex. specialized routing table management) cannot be maintained by StreamCode. To support more flexible active L3 functionality, only one execution environment (i.e. StreamCode Processor) is not enough for an active node, and another execution environment for the above long-lived algorithm supporting active L3 functionality is required.

Taking into account the above requirements, our StreamCode based active node is organized as illustrated in Figure 2. In this figure, bottom two boxes represent hardware resources. The left box represents hardware of each line interface including layer 1 and layer 2 interface, StreamCode Processor, packet buffer, and interface to packet switch. Right box represents a central hardware including node's main CPU, main memory and packet switch. The packet switch is passive backplane on which packets are exchanged between line interfaces. Each part of hardware provides software execution environment for active network programs. StreamCode Processor controls layer 1 and 2 line interface, buffers, and switch interface to provide StreamCode Processing Environment (SC-PE). SC-PE is an abstract environment for StreamCode program. Right side of the figure uses general active networking terms[1], and the concept is the same as other active networking systems, except that the memory access from SC-PE must be managed by node OS, as shown at the center of the figure. Concerning the interaction among two execution environment, we provide a mechanism to the StreamCode Processor's MMU to view a specific part of node processor's main memory. Although a straightforward definition of these interface may be (extendable) function call through the node OS to the specific application running on the node OS, we did not adopt these interface because of calling delay and uncertainness of its completion of execution in sub-millisecond. These memory areas have to be managed by node OS as an interface window to layer-3 code processor, and the access right should be well managed by the node OS and each MMU. These shared memory may be eventually allowed to write from the StreamCode Processor, but that situation will require complex distributed shared memory management system, so we choose to provide read only access, at this time. Therefore, there are no direct interface from the StreamCode Processor to

node OS, but the StreamCode can send information to the node OS by making and tossing a small packet to the node OS as usual packet transfer. A detailed mechanism for such protected access is discussed in the next section.

Table 1: StreamCode Instructions

Instruction	*Assembly code example*	*Meanings*
MOV1, MOV2, MOV4,..	MOV4 r(1) => r(2) MOV2 r(100) => m(r(2),r(4))	load and store primitives
ADD, SUB,..	ADD r(1), r(2) => r(3)	arithmetic/logical calculation
SKIP	SKIP EQ, r(1), r(2) -> label	conditional/unconditional jump
FIN	FIN	finish execution

Table 2: Co-processor Related Instructions

Instruction	*Assembly code example*	*Meanings*
FUNC	FUNC OUT_SINGLE, r(1), r(2), r(4),... => r(9)	invoke co-processor's function
WAIT	WAIT r(9)	check co-processor's status
FIN	FIN r(9)	wait co-processor, then finish

The rest of this section, we focus on the design of StreamCode. Our system is mainly committed for hardware execution, so code should be executed by hardware processor directly. This leads that a code set of the processor must be standardized, and should be carefully designed. Generality of a code set, compactness of written codes, execution speed, and the complexity of processor must be balanced. Our first design follows RISC type instruction set which should provide proven performance. One typical feature of StreamCode is that we have introduced extensible instructions to invoke co-processor's function asynchronously. Routing table search function, and data path (inside the processor) control function are major functions implemented as co-processor. There is another way to define primitives by presuming target applications, then divide sequence to reusable functional components, like traditional intelligent network approach. We did not deploy this approach, because this approach limits future development of unpredictable, forthcoming applications. Primitive instructions are shown in table 1, and co-processor related instructions are listed in table 2. Actual bitmap assignments for each instruction code are still under optimizing phase, and not shown here.

Another feature of StreamCode specification is that we did not provide automatic code caching, nor external code fetching (i.e. from outside of node). If code caching is required by a StreamCode application programmer, then the programmer may explicitly write StreamCode program to load pre-stored StreamCode program fragment from on-node memory, on his/her own responsibility for loading delay and for existence of the program fragment on every StreamCode-capable nodes. This design eliminates code

finding/fetching mechanisms and its delay from StreamCode Processor's performance-critical part.

In current networking environment, application programmer can make applications behave contents sensitive, but network does not provide enough information and control to these applications. By introducing proposed StreamCode based active nodes, everything will change. All packet forwarding is under control of StreamCode inside each packet, and application programmer can change these in a StreamCode packet-by-packet basis. Furthermore, by using ordinary EE and interaction mechanisms between EE and SC-PE, more efficient and flexible networking environment can be achieved.

5. STREAMCODE PROCESSOR ARCHITECTURE

According to the StreamCode specification, we have designed a StreamCode Processor. Basically, primitive instructions are simple enough to be executed at one clock due to pipeline techniques. However there are some time consuming algorithms in a layer-3 functionality, like searching routing table, transferring packet contents to output interface, and calculating checksum, hash value, or CRC. To deal with this, co-processor is utilized and co-processor related instructions have been introduced. Based on these instructions, StreamCode Processor can invoke co-processor by specifying known co-processor identification number, with formulated number of arguments. Argument list is specified for each co-processor function, and invoking instruction has generalized argument passing mechanism for generality and future extendibility. Co-processor works in parallel with main processor, and returns a result value to the specified (as argument) register asynchronously. Main processor also has instructions like checking co-processor is still running or finished, blocking further execution and wait for co-processor to finish. Some of co-processor functions have to be standardized, and others are free to use. Standard ones include Content Addressable Memory (CAM) type table search, IPv4 and IPv6 types of routing table lookups, packet contents transfer engine. Non-standard co-processor may be implemented as node specific ASIC, or as configurable device like FPGA, and their existence can be located on network wide directory system as specialized resource functions. These co-processors may include media-specific transcoding processor, media-specific compression/decompression, and cryptographic engine.

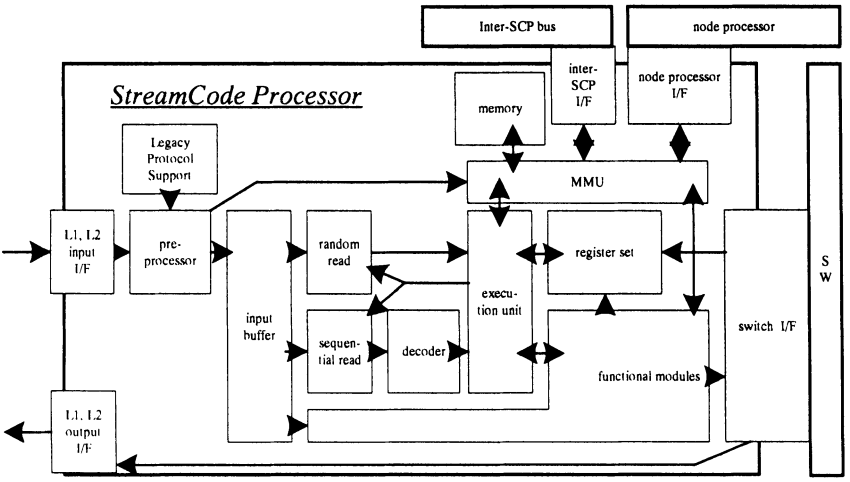

Figure 3: Line Interface Module with StreamCode Processor

Figure 3 shows functional modules of a line interface module, including StreamCode Processor, layer 1 and layer 2 input and output interfaces (L1, L2 input I/F and L1, L2 output I/F), switch interface, node processor interface and interconnection interface between StreamCode Processors on the same StreamCode node (inter-SCP I/F). L1, L2 input I/F analyzes input signal and assembles layer-3 packet according to physical/link layer protocols. The figure also includes packet switch (SW), node processor, and inter-SCP bus, which may be outside to the line interface module.

StreamCode Processor is constructed from functional modules described below:

- Input Buffer: The packet contents including StreamCode program are stored. This buffer has a valid pointer inside, and maintains how much contents are already stored from a L1/L2 input interface. If a sequential read function, a random read function, or one of functional modules try to access beyond that valid pointer, the access will be delayed until target region of the buffer will be filled by actual packet contents or StreamCode program. This blocking function will be reset if an arriving packet is shorter than expected, and at that time, whole execution is of course abandoned. As this figure shows, input buffer has one write interface and multiple read interfaces. These multiple access should be simultaneously allowed to avoid a pipeline stall situation. Size of input buffer follows expecting largest MTU size used in the network, and may be as 16K bytes to 32K bytes. Regarding to the current microprocessor's implementation, these size of cache can catch up processor's core clock without any delay, thus, this input buffer can be thought as the first level cache.

- Sequential Read Function: This reads StreamCode program from input buffer sequentially, and feeds instructions to a decoder. This can be thought as the internal (i.e. inside StreamCode Processor) instruction fetch unit. If branch has occurred, branch target should be set by execution unit, and this function will read the next instruction from that point. Many general processors have special functions with the instruction fetch unit to implement a speculative execution, expecting not making cache miss which make the total execution be slow. StreamCode Processor does not have so many disadvantages even if branch has occurred unexpectedly. Because all instructions are already on input buffer, or, sequential read function has no ways for waiting these pieces of StreamCode program if requested instructions are still on the wire and cannot be fetched.
- Decoder: This reads the output of the sequential read function, and decodes it as StreamCode instructions'. Finally feeds this analyzed information to the execution unit.
- Random Read Function: This is driven by the execution unit to read random place of input buffer. An access method is the same as the sequential read function except that reading target memory can be randomly specified.
- Register Set: Group of registers. Before execution unit starts processing an incoming new StreamCode program, all registers are reset to zero. In StreamCode Processor, the memory access is relatively slow because of its protection mechanism, and the amount of memory is limited for temporal use of each StreamCode program execution. To help StreamCode program run faster, we provide 256 registers, at an architecture level. Each register has currently 32 bit width, and serves free read and write operations to execution unit. Additionally, each register has an extra synchronizing support bit outside 32 bits, which reflect whether co-processor has finished the job or not. This bit is set as "not finished state", when the register is designated as storing a space of co-processor execution. Accessing to a register which has a synchronizing bit set as not finished, will cause the read access to block, and after the completion of co-processor's execution, the read operation continues.
- MMU: Every access to memory requires two parameters; one for specifying the memory space identification number, and another for offset inside that memory space. Memory spaces can be obtained through MMU requested by the node OS and that request may be invoked by EE application. When the node OS allocates memory area, it generates an authentication key value and sets it to MMU, and returns that information to the requesting EE application. Later, the EE application sends the key information to an end user's application, and the application sets the key information to StreamCode programs to use that memory space. The

actual memory space identification number is a StreamCode Processor dependent value, and not usable on other StreamCode Processors. It is an unrealistic idea to carry every memory identification number in a StreamCode packet; we make one level indirect mapping, or translation table, of the memory space identification number. At first, network wide application generates the pseudo number which may be identical to that one time, then use EE applications to allocate memory spaces with the generated number as a tag number. Each node OS on network nodes will allocate requested memory, and store a key-value pair to MMU, the tag number as a key, and the physical memory area identification number as a value. After the allocation, StreamCode program can access memory area by showing the tag number, and then get the memory identification number, and StreamCode program finally asks MMU to open access to that memory area. The authentication key value can be the same value all over the network, if EE application is written to do so. Another implementation may store the authentication key of the next node, inside the locked memory area. In that case, StreamCode program will be written as follows: open the lock, read key for the next node, replace packet contents to record that key, then go to the next node.

- Memory: This is a passive storage, managed by MMU. It can be implemented as uniform memory spool, but we have designed memory to have hint information, when it is allocated. We suppose hint can take value like 'temporal small buffers, disposable,' 'packet buffers to remember whole packet,' and so on. StreamCode Processor can be implemented to utilize these hint values in order to allocate disposable region on memory with 'clear all' functions, and to allocate packet buffers on memory with sequential read/write interface.
- Execution Unit: This controls entire execution of the processor according to a decoded StreamCode program sequence. This unit contains Arithmetic and Logical Unit (ALU) which performs calculation between registers, and short size data move (currently, 1 to 4 octets at once) between registers, from memory to register, or from register to memory. If an instruction requires access to memory, this unit checks the operand's memory space index number, next, if it is zero (it means that input packet should be read), this unit will trigger random read function with the specified offset. If memory space index is not zero, then the execution unit will access to MMU. This execution unit also controls invocation of co-processor, shown as functional modules. If some exception (ex. memory access violation, illegal instruction, etc.) is noticed by this unit, then total execution for the packet is simply abandoned. Implementing exception-handling mechanism is our future work.
- Functional Modules: These are functional logics of co-processors. These provide time consuming or complex functionalities to StreamCode

programs. A 'FUNC' instruction can be used to invoke each co-processor, and this instruction supports variable length argument list. All co-processors' argument information are registered at the execution unit, and when FUNC instruction is executed, their arguments are checked at run time.

- Pre-Processor: This checks layer 1 and layer 2 information of each incoming packet, and expects pre-configured routine jobs. Their jobs include invoking legacy protocol support function, reset processor after the end of packet detection, and fill input buffer by incoming packet contents.
- Legacy Protocol Support Function: This is invoked by a pre-processor to support legacy, i.e. not StreamCode based, layer-3 protocols. Actually, this module is a dictionary of StreamCode programs, indexed by legacy layer-3 protocols, and the stored StreamCode programs can be used to emulate each legacy layer-3 protocol. Pre-processor inserts this emulation program to each legacy packet, and the packet is executed as the StreamCode based packet.

One of the most important features of StreamCode Processor is a resource protection mechanism that is mainly achieved by MMU. We will discuss the resource protection with its related functionality in the rest of this section.

It is not allowed for any active program to eat up processor's resource, run out of network resource, nor access to other service's information. StreamCode may come from unknown end user's application; strict protection mechanism should be implemented in the processor. Code verification techniques are often proposed in active network research area, but these methods are suitable for the limited number of code sender, implying exchanging and storing trust information which introduce delay and storage scalability problems with our on-the-fly StreamCode execution system. We adopt rather hardware-oriented approach, again. StreamCode executions are forcibly terminated just after the final bit of the packet, which contains the executed StreamCode, is ready to read at the input buffer of the processor. StreamCode has to deal with this limit, or packet will be lost. Registers are always reset before new packet arrival, and memory can be accessed only through memory management unit (MMU), which requires some authorization key to open access rights. Each co-processor invocation is also managed by hardware, and has a default value for a number of invocations. To break these limits, authorization key is requested also. Furthermore, arguments to co-processor function, like routing table identification with table search function, output interface identification with packet output function, are also restricted by default. Some of them have

free access, and others not. For example, default IPv4 routing table and default IPv6 routing table may allow free access to all StreamCode programs, but other IPv4 routing tables specific to the VPN service, will request the authorization key to search that table. As a result, none of vicious StreamCode can break active L3 environment at a packet-by-packet execution level. At a network level, vicious StreamCode still have a method for denial of service attack, by looping around the network only using freely available resources. To deny that attack, disable free access for packet output co-processor function, or believe other network nodes that they honestly decreasing hop-count like resource usage statistics recorded on each packet.

6. PROTOTYPE SYSTEM

In this section, our proof-of-concept prototype system is discussed. The prototype includes software-implemented StreamCode interpreter, hardware-implemented StreamCode Processor built upon Field Programmable Gate Array (FPGA), and StreamCode over UDP/IP network, and sample applications running on this environment.

6.1 Software based StreamCode Processor

Before developing a hardware version of StreamCode Processor, we had built a software version of the processor to verify our concept of networking architecture. The software version additionally has a debugging functionality for StreamCode application programs. The software processor handles StreamCode over UDP/IP, by using specified UDP/IP connections as unreliable layer 2 links.

6.2 Hardware based StreamCode Processor

As our project aims to achieve hardware based active layer-3 functionality, hardware implementation feasibility must be demonstrated. We have developed FPGA based StreamCode Processor to show that. Brief implementation notes are described bellow.

Current implementation is at its very first step, and we intend to examine its usability, stability, and execution bottleneck of implemented logic. Due to its FPGA based implementation, we can easily change its hardware logic, to tune up our implementation, to test several ideas, and to improve our architecture.

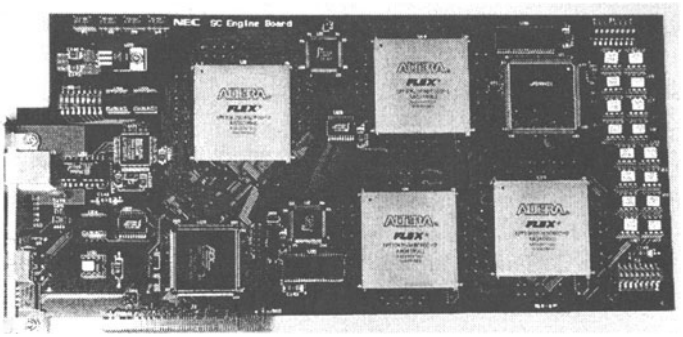

Figure 4: Prototype StreamCode Processor Card

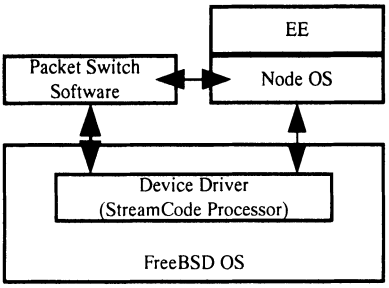

Figure 5: Supporting Software modules

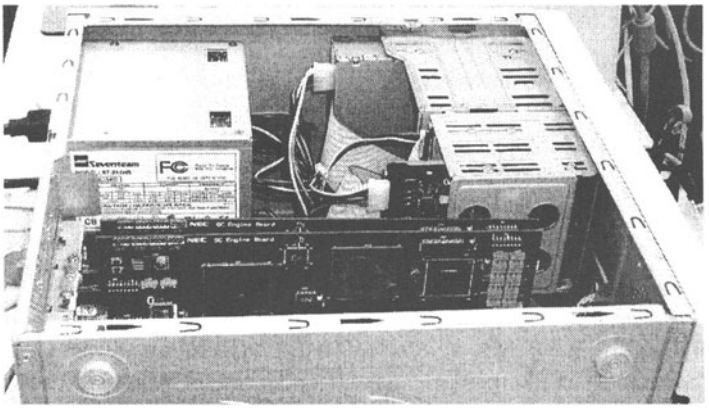

Figure 6: Mounted Processor Card

- Processor Card: We have implemented our processor as PCI card, and one PCI card works as one StreamCode Processor equipped with line interface. Figure 4 shows the appearance of the card, and Figure 5 describes software components on FreeBSD supporting these cards.

Multiple cards can be mounted to a usual personal computer as shown in Figure 6. An application software running on EE can control each card and packet switch software through the node OS. An incoming StreamCode based packet is processed by hardware StreamCode Processor, then, as a result of executing StreamCode program, the packet (may have some modifications by its StreamCode execution) is transferred to software packet switch with desired the destination interface identify number. The software packet switch will send each packet to the specified interface.

- Processor Core: Basic execution units of processor core are implemented on FPGA, including input buffer control unit (fill it, read it sequentially or randomly), decode unit, execute unit, co-processor interface, and registers. Input buffer is implemented by dual-port SRAM chip, and memory unit that can be accessed from execution unit, is implemented by SRAM chip. First implementation had dropped legacy protocol support unit, MMU, and pipelined execution.
- Co-Processors: We have implemented one co-processor, multipurpose data copy function. This function unit is implemented on FPGA, and can be used as FUNC OUT_SINGLE instruction. This unit can access input buffer, memory, register, and main memory of personal computer. Main memory is accessed to transfer packet content to/from the software packet switch. We prepared a space to mount our routing table search chip [12], which provides the longest prefix match search function in less than 100 nanoseconds (10 million search/sec).
- Outer Interface: There are two outer interfaces in this card, 100Mbit/sec Ethernet interface and PCI interface. Layer 1 handling of Ethernet packet is done by a dedicated chip, and layer 2 function is implemented on FPGA, which is now configured to handle StreamCode over UDP/IP. Hardware based processor and software based processor use the same packet format, so they are interoperable and can treat the same StreamCode packet. PCI interface is also implemented as a dedicated PCI bridge chip and some control functions on FPGA logic. DMA can be used between FPGA logic and the main memory of a personal computer, through the PCI interface chip.

6.3 Sample Applications

We have developed two sample StreamCode applications to check processors and the concept of StreamCode based active networking architecture.

First application is simple IPv4 like packet forwarding application. The code was written as to search destination address by table search function,

then decrement hop count and forward entire packet to the output interface according to the search result. StreamCode object for this application occupies 106 octets of each packet.

Second application is multi-QoS multicast application. For this application, a multicast user table is prepared on memory area of each StreamCode Processor. A server machine sends multiple MPEG streams simultaneously, each stream contains the same contents but with different quality (one packet contains no more than one quality). All streams are packetized with StreamCode object and traveling through network belonging to the prepared multicast table. The table also contains maximum quality levels for each destination interface, which is calculated by end user's and network administrator's request separately. On each node, if a stream has higher quality than requested, or if the destination interface is congested to satisfy the quality, this stream will be dropped. On a node just before the receiving terminal, only one stream with the highest quality, among available streams that have a right to the final destination, is selected and forwarded to the terminal. The server machine attaches a different StreamCode object to each packet, according to a target picture's quality, importance, and contents meaning. The difference of each StreamCode object will play different behavior of each packet, like how congestion level is seriously evaluated for the packet/picture, which (user's or network administrator's) request is precedent to others, etc. All this functionality is written in less than 400 octets StreamCode object.

7. SUMMARY

We proposed architecture and its first implementation of hardware based active layer-3 networking system, which consists of object code specification of StreamCode, StreamCode Processor, and active network nodes. Our implementation is still in progress, however, examined applications validate our system's ability to execute complex application using StreamCode Processors. While our FPGA based StreamCode Processor currently runs at only 16MHz clock with its inefficient implementation, it can read packets with short benchmarking StreamCode object at 90Mbits/sec speed, and handles complex multi quality multicast application at 10Mbits/sec speed.

We are currently redesigning StreamCode instruction code to shrink the size of object code and remove bottleneck of execution, and improving FPGA implementation for completing its full functions. As an interim result of our efforts, now the processor core's performance for code fetching and

execution is estimated at 22.5MIPS / 650Mbps (average score) with 27MHz processor-clock (FPGA-simulator's estimate). This result shows that for 100Mbps link, even in hardest situation like packets fully-filled with StreamCode program are arriving at full link speed, the processor can execute StreamCode programs in each packets 6 times (as iteration) within the packet's lifetime. Next, we have to develop programmable packet-by-packet buffer management mechanisms, i.e. QoS control system suitable for our platform, where execution effectiveness, functional flexibility and network-wide stability must be balanced. At the same time, we should refine protocol stack / middleware on terminals, to provide fully activated layer-3 network functions to end user's applications. Application programmer's commitment to active network is a key to promise a future networking environment.

REFERENCE:

[1] Calvert, K., "Architectural Framework for Active Networks Version 1.0," http://www.dcs.uky.edu/~calvert/arch-1-0.ps , July 1999

[2] Yemini, Y., et al., "Towards Programmable Networks," in Proc. NOMS 96, October 1996.

[3] Tennenhouse, D., et al., "A Survey of Active Network Research", IEEE Comm. Mag., January 1997.

[4] Wetherall, D., et al., "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," in Proc. IEEE OPENARCH'98, April 1998.

[5] Alexander, S. et al., "The SwitchWare Active Network Architecture," IEEE Network Mag., pp. 29-36, May/June 1998.

[6] Braden, B., Ed., et. al., "Resource Reservation Protocol (RSVP) - Version 1 Functional Specification", RFC 2205, September 1997.

[7] Deering, S., Ed., et al., "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, December 1998.

[8] Blake, S., Ed., et al., "An Architecture for Differentiated Services", RFC2475, December 1998.

[9] Campbell, A. T., et al., "The Genesis Kernel: A Virtual Network Operating System for Spawning Network Architectures," in Proc. IEEE OPENARCH'99, March 1999.

[10] Nygren, E L., et al., "PAN: A High-Performance Active Network Node Supporting Multiple Mobile Code Systems," in Proc. IEEE OPENARCH'99, March 1999.

[11] Decasper, D., et al.,"A Scalable, High Performance Active Network Node," In IEEE Network, January/February 1999

[12] Kobayashi, M., et al, "A Longest Prefix Match Search Engine for Multi-gigabit IP Processing," IEEE ICC 2000, June 2000

[13] EZchip Technologies, "Network Processor Designs for Next-Generation networking Equipment," http://www.ezchip.com/, Dec 1999.

[14] Hadzic, I., et al., "On-the-fly Programmable Hardware for Networks," in Proc. IEEE Globecom 98, November 1998