**Communications and Network, 2015, 7, 55-69**
Published Online February 2015 in SciRes. http://www.scirp.org/journal/cn
http://dx.doi.org/10.4236/cn.2015.71006

Scientific Research Publishing

# High-Level Portable Programming Language for Optimized Memory Use of Network Processors

**Yasusi Kanada**

Central Research Laboratory, Hitachi, Ltd., Yokohama, Japan
Email: Yasusi.Kanada.yq@hitachi.com

Received 26 January 2015; accepted 15 February 2015; published 16 February 2015



## Abstract

**Network processors (NPs) are widely used for programmable and high-performance networks; however, the programs for NPs are less portable, the number of NP program developers is small, and the development cost is high. To solve these problems, this paper proposes an open, high-level, and portable programming language called "Phonepl", which is independent from vendor-specific proprietary hardware and software but can be translated into an NP program with high performance especially in the memory use. A common NP hardware feature is that a whole packet is stored in DRAM, but the header is cached in SRAM. Phonepl has a hardware-independent abstraction of this feature so that it allows programmers mostly unconscious of this hardware feature. To implement the abstraction, four representations of packet data type that cover all the packet operations (including substring, concatenation, input, and output) are introduced. Phonepl have been implemented on Octeon NPs used in plug-ins for a network-virtualization environment called the VNode Infrastructure, and several packet-handling programs were evaluated. As for the evaluation result, the conversion throughput is close to the wire rate, *i.e.*, 10 Gbps, and no packet loss (by cache miss) occurs when the packet size is 256 bytes or larger.**

## Keywords

**Network Processors, Portability, High-Level Language, Hardware Independence, Memory Usage, DRAM, SRAM, Network Virtualization**

## 1. Introduction

To enable programmability for networking and in-network processing, especially for new network-layer pro-

**How to cite this paper:** Kanada, Y. (2015) High-Level Portable Programming Language for Optimized Memory Use of Network Processors. *Communications and Network*, **7**, 55-69. http://dx.doi.org/10.4236/cn.2015.71006

gramming for clean-slate virtual-networks [1], network processors (NPs) have been used [2] and will be more widely used in the near future. NPs, which were developed for software-based high-performance networking solutions, make it possible to quickly develop arbitrary protocol and functions in the case of hardware-based solutions as well.

However, there are three problems that make using NPs for such functions difficult. The first problem is lack of portability. Because low-level languages that are similar to assembly languages must be used for developing NP programs, the programs are not portable. Although extended versions of C can usually be used for developing NP programs, essential libraries depend on vendor-specific proprietary hardware and software, and proprietary rights on NP programs are protected by non-disclosure agreements (NDAs) preventing programs and documents concerning an NP being ported. The second problem is high development cost and that the availability of NP program developers is limited. NP program developments require special skills, and the knowledge they require is not widely available; thus, only a limited number of developers have the ability to develop NP programs. In addition, vendor-specific information is required in NP-program development. Consequently, the learning curve of NP-program development is very gentle, the development takes a very long time, and its cost is very high. The third problem is restriction on publishing developed programs, papers, and documents concerning an NP. This is a serious problem for network researchers.

The three above-described problems can be solved by successfully designing and implementing a high-level-language, which can translate programs into NP machine code or a vendor-dependent C program. Programs written in this language must be translated into NP-dependent object programs; however, to solve the problems, the language must be hardware- and vendor-independent.

An important common NP feature concerning high-performance packet processing (to avoid packet drops caused by cache misses) is to use static random-access memory (SRAM) and dynamic random-access memory (DRAM) by different methods with explicit awareness by the programmer, making programming difficult and time-consuming. Although memory allocation is not the only issue that causes the above three problems, this is the most important and serious issue because NPs are optimized for wire-rate processing and memory abuse immediately prevents it and severely reduces the performance. In particular, whole packets are stored in DRAM, and only the headers, which must be modified, removed, or added, are cached in SRAM because if data stored in DRAM is accessed by a CPU core, access takes an excessively long time, and wire-rate processing is impossible. The rest of the packets are just forwarded to the next network node without modification in the NP. This is common because it is necessary for NPs to store packets in memory while processing them, but the size of SRAM is limited, so whole packets cannot be stored in short-access-time memory, *i.e.*, SRAM.

When programming a packet-processing program for NPs, programmers must use an assembly language or C with assembly-level features, and must be very careful to get high performance. When using general-purpose CPUs, programmers can use high-level language and do not have to distinguish SRAM (or cache) and DRAM because they are automatically selected when programs load and store data. However, NP programmers must usually know whether the packet to be processed is on SRAM or DRAM (or both) because this knowledge is critical for attaining stable (*i.e.*, mishit-less) wire-rate processing. Two types of NP architectures are available. In one of them, such as Intel IXP, the SRAM and the DRAM are different classes of memory with different addresses. In the other type, such as Cavium Octeon®, the SRAM can be accessed as cache or registers, in a similar manner to general-purpose CPUs, but programmers must still be aware of the SRAM/DRAM distinction because the NP handles them in different ways. These cases are explained in more detail in Section 2.

Although it is a promising approach to design a new open and portable high-level language and to implement a high-performance language processor, *i.e.*, a compiler and run-time routines, it is still very hard to solve the above three problems because of the wide semantic gap between the language and the object program.

However, this paper describes the successful first step toward this goal. Hardware features such as those described above can be abstracted to common high-level language features that do not make programmers conscious of the low-level hardware features. To enable this type of abstraction, a high-level language called “Phonepl” (portable high-level open network processing language) is proposed, and a method for compiling packet-handling programs in Phonepl into high-performance programs that can fully utilize hardware while distinguishing SRAM and DRAM is proposed. Here, “open” means that network processors can be programmed without NDAs. Especially, packet headers are automatically cached, the language processor is aware that the data being handled is stored in either SRAM or DRAM (or both) and manages data transmission between them, and programmers do not have to pay attention to this distinction, so the programming cost can be decreased.

Phonepl does not depend on vendor-specific NP hardware and software, and thus the programs in Phonepl can be portable among various NPs.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 describes Phonepl. Section 4 describes a method for implementing Phonepl for NPs, especially four representations of packet type and a method for handling them. Section 5 describes a prototype implementation of Phonepl for plug-ins for a network-virtualization environment called the VNode Infrastructure, and Section 6 evaluates it by using several applications. Section 7 concludes this paper.

## 2. Related Work

This section focuses on previous studies on NPs and languages for packet processing because, although there are many studies on memory-related optimizations concerning high-performance computing, such as Sequoia [3], they focus on array processing and the requirements for packet-stream processing are quite different from them.

### 2.1. Selection of SRAM/DRAM in NPs

The IXP series of NPs developed by Intel [4] does not have cache, and its SRAM and DRAM have different memory spaces. The developers at Intel reported that cache is not effective in the case of NPs, so this type of memory architecture is good for network processing. However, it is difficult to program IXP processors because programmers, who are not even aware of the difference between SRAM and DRAM, must use them with different methods.

In contrast, the architectures of NPs developed later, for example, Cavium Octeon® [5] and Tilera® Tile Processors [6], are more similar to those of general-purpose CPUs with cache. However, because a cache miss may disable wire-rate transmission of packets, there are several devices that can be applied to avoid cache miss. That is, data to be processed at wire rate must be stored in SRAM. However, because an NP cannot usually have sufficient quantity of SRAM to store all the processing packets, it only stores descriptors and headers of packets in SRAM, and the rest or whole packets must be stored in DRAM. Various different types of packet-processing hardware and software behave in a similar way. In addition, to process packets at wire rate, NPs distribute packets to many cores for parallel processing, and they sort the resulting packets by hardware in input order and queue them for output or the next processing.

### 2.2. Selection of SRAM/DRAM with a Packet-Processing Language

In an NP program-development environment called Shangri-La [7], which was developed by Intel and several universities, a high-level language called Baker [4] was developed for IXP. By assuming that packet bodies are stored in DRAM and descriptors are stored in SRAM, Baker enabled programmers to handle packet data without having to consider whether they are on DRAM or SRAM. The data structure on SRAM, however, must be designed by programmers, so it depends on NP architecture. In addition, programmers must describe data transmission between DRAM and SRAM, so they must explicitly describe caching operations.

Unlike Octeon or Tilera, Baker does not have a mechanism for supporting automatic distinguished use of SRAM and DRAM. It is therefore difficult to process packets at wire rate by using Baker.

### 2.3. Packet-Stream and Data-Stream Languages

Click [8] is software architecture for describing routers modularly. Two-level description is used in Click. The lower level, or component level, is described by C, and the higher level is described by a domain-specific language, which connects modules in several ways. Click programs can be portable, but it is difficult to get high performance from portable Click programs. NP-Click [2] is a specialized implementation of Click for IXP NPs. Modules in NP-Click are written in IXP-specific C language; there-fore, the programs are not portable.

Frenetic [9] is a language for controlling a collection of OpenFlow [10] switches. It is embedded in Python but is based on SQL. It is a declarative language and processes collections (streams) of packets instead of processing individual packets procedurally in the manner of Phonepl. Because Frenetic processes packet streams, it is very similar to CQL (Continuous Query Language) [11]. Unlike Phonepl, Frenetic can only be used to program the control plane; it cannot handle the data plane.

NetCore [12] is a rule-based language for controlling OpenFlow switches. Rules in NetCore are condition-

action rules; that is, rules that match incoming packets are activated.

## 3. Packet-Processing Language

A high-level language called Phonepl, which solves the three problems described in the introduction, is outlined.

### 3.1. Basic Design of Phonepl

Phonepl is designed for wire-rate (low-level) packet-processing of any format, such as a non-IP and/or non-Ethernet format, as well as designed to be as close as a conventional programming language, *i.e.*, Java, because it should be easy to handle by Java and C++ programmers.

The reason why a new language, which is open, portable, and easy to use, is designed is explained as follows. Although it is close to conventional languages, a new language is required because it is very hard to compile a general-purpose program to high-performance object program for NPs, which very optimized hardware-usage, especially memory usage, is required for. Phonepl may thus be considered as a very restricted and extended version of Java.

Two major design goals of Phonepl are as follows. First, Phonepl must be high-level; that is, it must be designed for the programmer not to be aware of proprietary hardware and software. Second, Phonepl must be able to express high-performance packet-processing programs. Especially, processing at wire-rate, *i.e.*, 10 Gbps or more, without packet drops is required. In an NP, input packets may be partially cached, that is, the header of the packet is stored in SRAM and the rest or whole packet is stored in DRAM, but a DRAM access may disable wire-rate processing and cache miss easily cause packet-drops. However, this goal must be achieved without abandoning the first goal, *i.e.*, high-level programmability.

To achieve these design goals, data structures, especially Packet and String, which are the most important data structures in Phonepl, must be carefully designed and the method for processing them must be developed. Especially, packets are designed to be immutable byte strings in Phonepl and they are distinguished from non-packet strings.

There are five language features concerning this design. The first feature is that packets are byte strings because packets with arbitrary formats should be able to be handled in uniform methods. Packets have variable length, so they can be handled as byte strings (similar to character strings). A packet in Phonepl is not a encapsulated object. This decision makes low-level and cross-layer optimization of packets easier. The protocol-handling method written in Phonepl is thus completely different from that written in Java.

The second feature is that packets are immutable. Packets are handled as immutable (non-rewritable) objects, which are similar to character strings in Java or other languages; that is, packet contents cannot be rewritten. This immutability enables memory areas, especially DRAM areas, to be shared by packets before and after an operation.

The third feature is that types of packets, *i.e.*, Packet, and non-packet strings, *i.e.*, String, are different in Phonepl. They are incompatible for two reasons. First, although they can be logically identical, they must be implemented by using quite different methods and this distinction makes implementation more efficient and easier. Operations such as subpacket and substring described below utilize this difference. Second, programmers can easily distinguish them. Non-packet strings are used for temporary data, e.g., packet fragments, but packets are used for I/O data; that is, packets and packet fragments (non-packets) are different for programmers.

Two assumptions are made in regard to implementation of these data types. The first assumption is that whole String objects are stored in cacheable memory, *i.e.*, in SRAM, but can be stored in DRAM if needed. If they are in cache, purging the cache may have to be inhibited. The second assumption is that only the head of a packet is cached, and the tail is stored only in DRAM. However, a short packet may be wholly cached and may be stored only in SRAM.

The fourth feature is that packet and non-packet byte-substring operations are different in Phonepl because the types of the operation results are different. A new packet can be generated by removing part of another packet using a subpacket operation, and a non-packet byte string can be generated by extracting part of a packet using a substring operation. These operations can have the same name *i.e.*, a substring, but are distinguished.

The fifth feature is that packet- and byte-*concatenation* operations are specialized. A byte string can be generated by concatenating two or more byte strings by a concat operation, and a packet can be generated by concatenating one or more byte strings and a packet by a packet constructor called “new Packet”. Although a packet

can logically be generated by concatenating multiple packets, such concatenation seems to be practically less useful and difficult to implement, so no such operation is included (See Section 4.2.3 for more explanations).

## 3.2. Program Example and Packet Operations

To outline Phonepl and to explain several data structures and important packet operations, a program that performs MAC-header addition/removal, which cannot be performed by conventional non-programmable network nodes, is shown in **Figure 1**. The program in this figure defines class AddRemMAC. It has two functions that handle two bidirectional packet streams, *i.e.*, NetStream1 and NetStream2 (lines 001 - 002), which are bound to physical network interfaces outside this program. One function inputs packets from NetStream1, generates new packets with a new MAC header (*i.e.*, adds a new MAC header at the front) for each packet, and outputs them to NetStream2. The other function inputs packets from NetStream2, removes the MAC header in front, and outputs it to NetStream1. The program is much simplified because it is sufficient to show the functionality and basic implementation of the language; that is, no validation test is performed before the header is added or removed. However, it is easy to add check code to this program.

Packet flows are handled as "streams" in Phonepl. Method of stream handling is described using the constructor of class AddRemMAC here. The parameter declarations of AddRemMAC (lines 006 - 007) specify that input packets to parameter port1 pass to method process1 and input packets to parameter port2 pass to method process2. This type of parameter declaration is Phonepl specific; that is, Java grammar is modified for the sake of stream processing. The parameter values (packet streams) are assigned to instance variables out1 and out2 to make them available in the newly created object. Methods process1 and process2 receive one packet at a time. (One of these methods is executed once on only one core for each packet.) Because Phonepl handles input packets by these methods only, there is no specific method or statement for packet input.

Examples of a substring operation (which is used for accessing packet components), a packet constructor (which is used for packet composition), and a packet-stream output using "put" method can be seen in method

```
001  import NetStream1;
002  import NetStream2;

003  class AddRemMAC {
004    NetStream out1;
005    NetStream out2;

006    public AddRemMAC(NetStream port1 > process1,
007                     NetStream port2 > process2 ){
008      out1 = port1;
009      out2 = port2;
010    }

011    void process1(Packet i) {
                      //Port 1 to 2 (no VLAN -> no VLAN)
012      Packet o = new Packet(i.substring(0,14),i);
           //  MAC header of original packet (i: Original packet)
013      out2.put(o);
014    }

015    void process2(Packet i) {
                      // Port 2 to 1 (no VLAN -> no VLAN)
016      Packet o = i.subpacket(14);
           // remove MAC header (no VLAN)
017      out1.put(o);
018    }

019    void main() {
020      new AddRemMAC(new NetStream1(),
021                    new NetStream2());
022    }
023  }
```

**Figure 1.** Simple MAC-header addition/removal program.

process1 (line 011). This method handles a packet that comes from NetStream1, generates a byte string from the first 14 bytes of input packet i (it is assumed that the size of MAC header is 14 bytes) by i.substring(0,14), generates a packet by concatenating this byte string and the original packet by new Packet(⋯,i), and outputs the resulting packet to NetStream2(out2).

An example of subpacket operation, which generates packets from an existing packet, can be seen in method process2 (line 015). This method handles a packet that comes from NetStream2, generates a packet by removing the first 14 bytes of input packet i by i.subpacket(14), and outputs the resulting packet to NetStream1 (out1).

Finally, an example of stream initialization is seen in function main() (line 019). When class AddRemMAC is initialized, this function is executed. It logically runs only once, but each processor core may execute it once unless there are side-effects. It generates an instance (a singleton) of class AddRemMAC, which runs forever and processes packets repeatedly unless it is externally terminated. Two packet streams are generated and passed as arguments of AddRemMAC. They start to operate (input and/or output packets) when instances are generated.

## 4. Implementation Method

To implement semantics close to conventional programming languages such as Java, a special method of handling data (object) is required for Phonepl. The key feature of Phonepl implementation is the four representations of packets and operations among them.

### 4.1. Four Representations of Packets

In Phonepl, multiple packet data-representations used in NPs are unified as a single data type called Packet. Four different representations shown in **Figure 2(a)** (explained below) are therefore used for Packet. These representations are required because of the following two reasons concerning high-performance packet-processing and NP hardware. First, in most packet-processing in network nodes, packet headers are added, removed, or updated, but packet tails, *i.e.*, payloads, are not touched unless very deep packet-inspection is required. So the packet headers must be stored in SRAM (or scratchpad memory) but the packet tails can be stored in DRAM as described in the introduction and in the previous section. It is usually not possible to cache whole packet. Second, NPs are designed to handle input and/or output packets by specialized hardware. The hardware is optimized for the packet-processing requirements described above, but some hardware-specific restrictions apply in addition.

An example of hardware-specific data representation that matches the abstract representation is shown here. In some NPs, there are input-specific and output-specific packet formats using a special descriptor format. Short packets may be fully stored in SRAM but packet heads may be stored in both SRAM and DRAM for longer packets. The four abstract representations are designed to generalize various concrete representations, such as shown in **Figure 2(b)**, used in NPs. Although the descriptor format is specialized, it can be abstracted as shown in **Figure 2(a)**. If vendor-specific C language is used, these representations are handled separately; however, Phonepl, handles them uniformly. Even for cases that the NP has a cache, it is probably useful to distinguish multiple representations because cache miss must be avoided.

The four representations are explained in the following.

- **Cached:** The whole packet data is stored in SRAM. It is not assumed that a copy of the data is stored in DRAM.
- **Mixed:** The head of a packet (the number of bytes depends on implementation) is stored in SRAM, and whole packet data is stored in DRAM.
- **Gathered:** A packet consists of multiple fragments. Each fragment is stored in a memory area (*i.e.*, DRAM or SRAM). A gathered packet can be represented by an array or a linked list of fragments.
- **Uncached:** The whole packet is stored in DRAM. It is not assumed that a copy of the data is stored in SRAM.

Packets inputted to NPs are usually in cached or mixed representation; that is, short packets may be represented by cached representation but mixed representation is required for long packets. All four representations are used for expressing operation results and may be used for output. However, reasoning of mixed, gathered, and uncached representations are explained more.

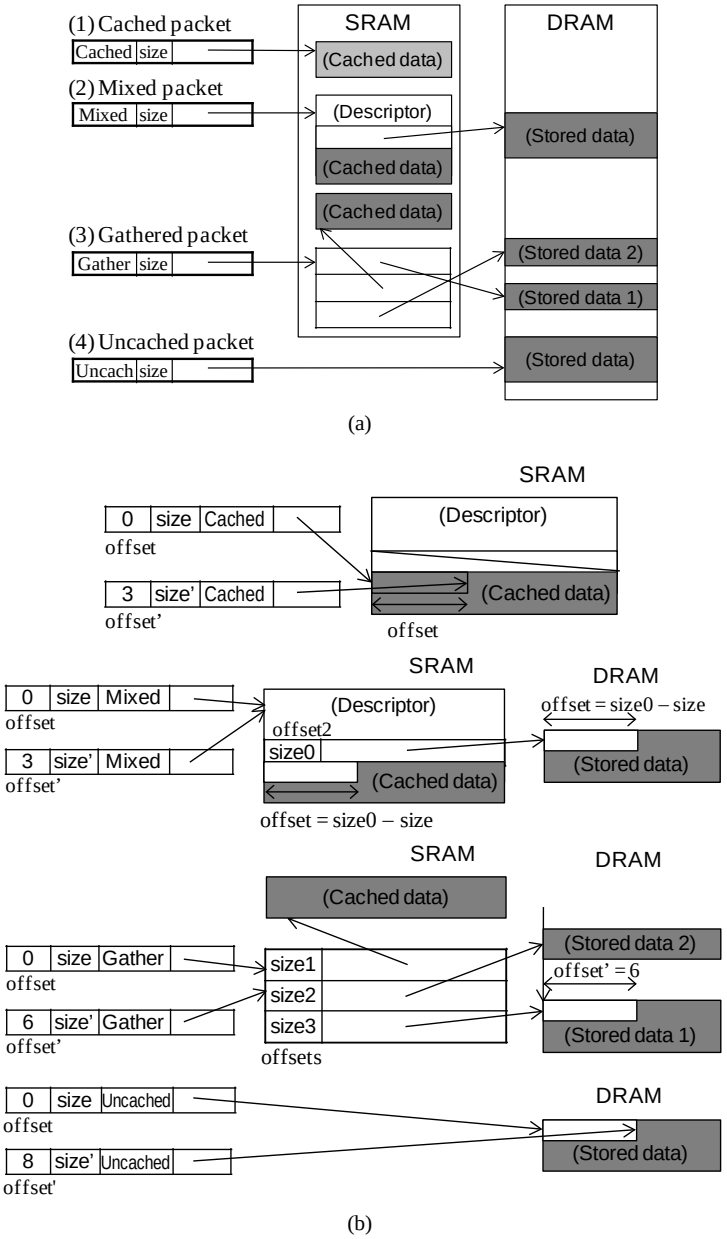


**Figure 2.** Four representations of packet type. (a) Abstract representations; (b) Examples of more detailed representations.

Mixed representation is required because, in packet processing, only the packet head (containing headers) is usually modified, headers are added or deleted, and the packet tail is kept unchanged. Good performance can therefore be obtained by caching only the head to SRAM and storing the tail only in DRAM. Data accessed by cores must be stored in SRAM because if data stored in DRAM is accessed, it takes excessively long time, and wire-rate processing becomes impossible.

Gathered representation is required when generating a packet from multiple pieces of data stored in DRAM or SRAM. In such a case, if all the pieces are copied to a contiguous area (of DRAM), copy from DRAM to DRAM is required and wire-rate processing becomes impossible. This representation is closely related to the immutability of packets, which enables sharing part of a string.

Uncached representation is required when a packet is generated from a tail of another packet with gathered representation by an operation such as a header deletion.

Because the four representations may have to be distinguished at run time, a tag must be supplied. The tags should be in packet-data pointers. However, because packet data are handled by hardware in NPs, the data representation and handling methods in the case of a high-level language must be very carefully designed and implemented. If the address space is sufficiently large, a part of the address can be used for a tag. This representation is close to widely used methods for dynamically-typed languages, such as Python or Lisp.

## 4.2. Packet Operations and Four Representations

Because there are four packet-data representations and each packet data has a tag, packet operations must be implemented for all these representations, and sometimes run-time tag check is required.

### 4.2.1. Run-Time Tag Check

Because there are multiple representations in Packet type, they must be distinguished dynamically (by the run-time routines in the NP) or statically (by the Phonepl compiler). In terms of efficiency, it is better for the representation to be statically distinguished. However, it is impossible to distinguish every representation of a packet statically, so run-time tag-check is, at least sometimes, necessary. Especially, if a non-optimizing compiler is used, tag check is always necessary at run time. Such a run-time check causes overhead, but it does not usually prevent wire-rate processing because the tags are in cached pointers and a tag can be added and removed with very small cost.

### 4.2.2. Packet I/O

Some NP hardware creates a descriptor when receiving a packet. The descriptor is in SRAM, and whole packet data may be stored in DRAM. The input packet format, thus, is close to mixed representation (or cached representation in the case of a short packet); however, a tag must be added when run-time tag-check is required. The run-time routine should thus decide which representation is to be used and insert the tag value. This means that the language processor must fill the gap (*i.e.*, convert) between data representations in the hardware and in Phonepl. If the gap is wide, significant CPU time is required to fill it, and performance may decrease. An appropriate representation design is therefore important.

An output packet format must be prepared for some NPs when sending a packet. One of the four representations should be close to the output format; however, the tag must be removed before passing the data to the packet output hardware. For example, the output format may be close to gathered representation, but the tag value "gathered" must be cleared. The hardware concatenates the fragments pointed to by the gathered representation and outputs the result.

### 4.2.3. Subpacket

Each representation requires different implementations of an operation to achieve a subpacket operation. In all the cases described below, the operations are executed using data stored in SRAM, and DRAM is not accessed.

If the packet has a cached representation, a subpacket of the packet is in a cached format. The original packet can be stored in the allocated SRAM area. The resulting subpacket may share the original packet data or may be a copy of the original data. In this case, because both the original and copied data are stored in SRAM, this copy operation probably does not prevent wire-rate processing.

If the packet has a mixed representation, a subpacket of the packet may be in a mixed or uncached format. That is, there are two cases. Firstly, if the resulting packet contains both head data stored in SRAM and tail data stored in DRAM, the result is mixed format. Secondly, if the resulting packet only contains tail data, the result is uncached format. In general, the resulting representation is not known at compile time because the range specified in subpacket operation might not be known at compile time. In both cases, a new descriptor is generated in SRAM by using the original descriptor, but no packet data stored in DRAM is accessed.

If the packet has a gathered representation, a substring of the packet is usually in a gathered format. The original and resulting packets may share the array of fragments (*i.e.*, only a packet-type pointer is generated) or the resulting pointer may point to a new array copied from the original array. An array copy probably does not prevent wire-rate processing because both arrays are stored in SRAM.

If the packet has an uncached representation, a substring of the packet is in an uncached format. Both the original and resulting packet data are stored in DRAM and shared. The address and the length of the resulting

packet are stored in a packet-type pointer. No packet data stored in DRAM are accessed.

#### 4.2.4. Concatenation

When a packet is generated by concatenating one or more byte strings (such as new headers and a packet content), a constructor, "new Packet()", is used. In the current implementation method, this constructor generates a gathered-format packet. That means, the parameter values of the constructors are the elements of the array in the gathered format. However, a more optimized method, which uses other representations, may be developed.

The last element of the constructor may be a packet of any representation. If this element has a mixed format, the DRAM part (which represents the whole packet) becomes an element of the array. If this element has a gathered format, each input array element becomes an element of the array of the output gathered format.

#### 4.2.5. Generating Packet without Using Input Packet

A packet can be created without using a pre-existing packet by using a packet constructor. The generated packet is in cached or gathered format. If the constructor has only one argument that contains a byte string, the resulting packet is in cached format, and if it has two or more arguments, the resulting packet is in gathered format.

### 4.3. Several Miscellaneous Issues

Two issues related to the proposed packet-handling method are explained in the following. The first issue is memory deallocation. Sharing part of packets and strings makes memory deallocation difficult. Garbage collection or reference counting can solve this problem completely, but the overhead is large. In the current implementation, strings that are (potentially) assigned to global (instance) variables are not deallocated. However, the current deallocation policy may cause memory leak. A more precise method should be devised in future work.

The second issue is adaptation to hardware-based memory allocation. Some NPs allocate and deallocate packet memory automatically to avoid software-memory-management overhead. When a packet arrives, the SRAM and DRAM required for the packet is allocated. However, it is difficult for NP hardware to decide when the packet memory can be deallocated. A Phonepl compiler must therefore generate code for deallocate it.

## 5. Prototyping

The above-described implementation method has been applied to a programming environment called +Net, which contains a Phonepl processor called +Net Phonepl. +Net Phonepl is used for programming physical nodes with a network-virtualization function and NPs.

### 5.1. Platform

The prototype compiles a Phonepl program and runs it on a "virtualization node" (VNode) [1] [13]. A virtualization platform called VNode Infrastructure, which supports multiple slices (*i.e.*, virtual networks) using a single network infrastructure, and a high-performance fully functional virtualization testbed were developed. The components of a VNode contain NPs. The prototype is a replacement of one or more NPs in this environment. The program has packet I/O streams as described in Section 3.

A source program is compiled according to the following procedure. First, an intermediate language program (ILP) is generated by using a Phonepl syntax/token analyzer. The syntax analyzer was generated using "Yet Another Perl Parser" (YAPP) compiler, which has similar functions as those of YACC (Yet Another Compiler Compiler) or Bison parser-generators but is written in and generates Perl code. The ILP is translated by using a Phonepl translator into a specialized C program. A GNU C compiler for Octeon compiles this C program and generates object code for an Octeon board called WANic-56512 developed by General Electric Company. A run-time library is linked to the object program. The main components of this library are an initializer, packet processors, and a packet-output routine.

### 5.2. Compiled Code of +Net-Phonepl Compiler

To outline the object-code structure and the compilation (or program transformation), an example of compiled code is explained here. The C program generated by the Phonepl compiler from the MAC-header addition/removal program (in **Figure 1**) is shown in **Figure 3**.

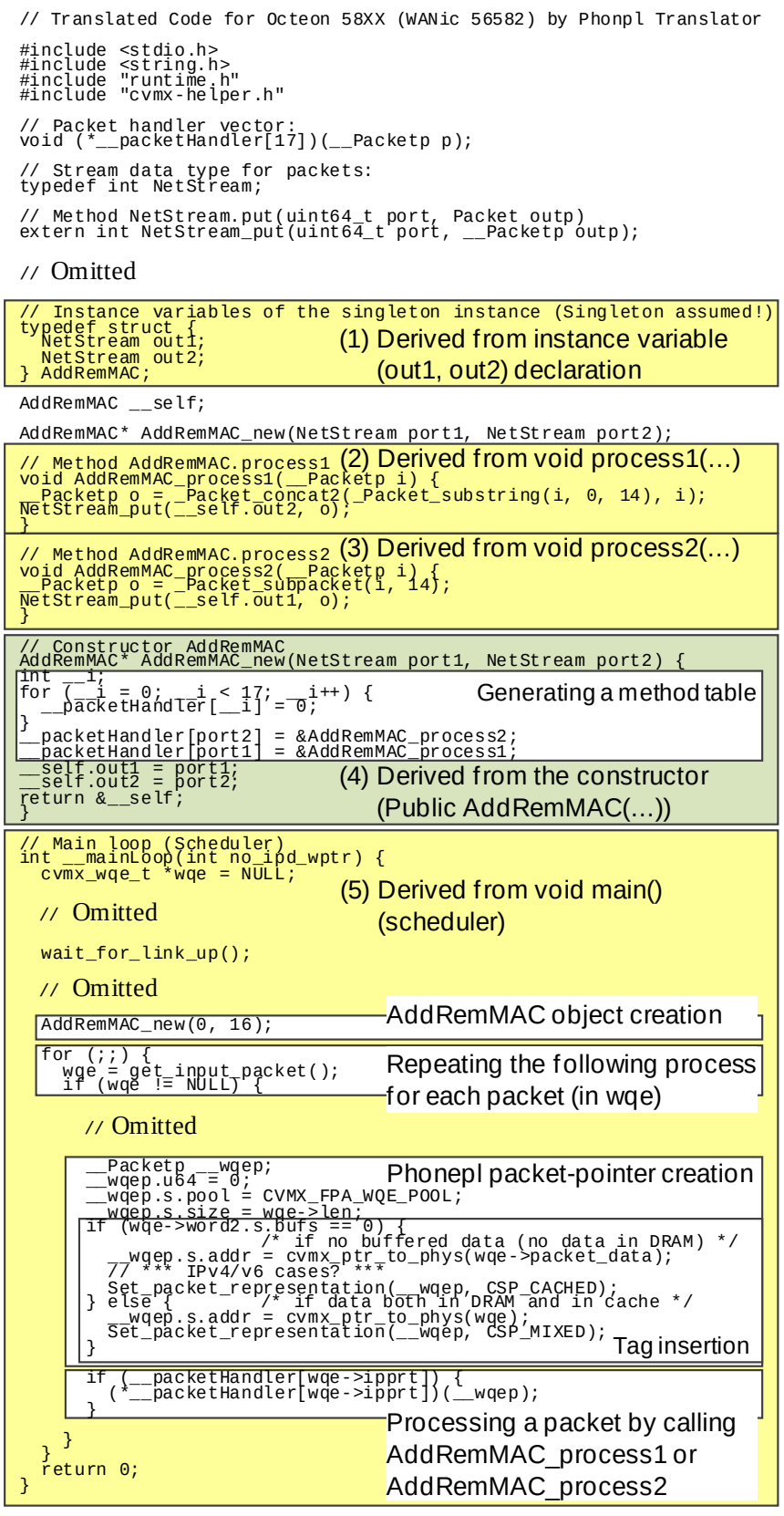


**Figure 3.** Compiled code of MAC-header insertion/deletion program.

This program is explained instead of describing detailed compilation process because the process is too much complicated and the program structure can probably be used for other types of NPs. A compilation technique specialized for a singleton (*i.e.*, single-instance class) is applied to this program. Cores in an Octeon processor execute this program in parallel; that is, each core processes a packet. The program consists of five parts: part 1 derived from instance-variable declaration, parts 2 and 3 derived from methods process1 and process2, part 4 derived from the constructor, and part 5 derived from the main program.

In part 1, AddRemMAC type, which corresponds to instance of class AddRemMAC in Phonepl, is declared. Because a compiled object of class AddRemMAC has two objects of NetStream type, the corresponding structure components are declared. In parts 2 and 3, *i.e.*, method definitions, the element names in the source program are replaced by the element names in the run-time library. The run-time routines may be expanded in-line; however, they are not expanded in this example program. In part 4, *i.e.*, the constructor of class AddRemMAC, methods process1 and process2 are initialized. Assignment statements that correspond to the assignment statements in the source program are included in this part. In part 5, *i.e.*, the main program, the above constructor is called, and every time it receives a packet, one of the above two methods are called. Because NetStream type is an abstraction of a packet stream, the stream elements are handled one by one, and the scheduler for this process occupies the main part of part 5. When the function get_input_packet() is called, a packet is received, and the data representation of this packet is converted to that of +Net Phonepl by adding a tag, *i.e.*, cached (CSP_CACHED) or mixed (CSP_MIXED).

## 6. Evaluation

Both the programmability, especially ease of language use, and the performance of the implementation should be evaluated; however, because Phonepl is being improved, performance is focused in this evaluation. Two Phonepl programs for network-layer packet handling were written. Prototypes with these object programs were used for extending VNode, and the traffic was measured.

### 6.1. MAC-Header Addition/Deletion Program

The first program performs MAC-header addition/removal. It is a modified version of the program shown in **Figure 1**, and similar programs are used for extending virtualization-node (VNode) functions by using the node plug-in architecture [14]-[16]. Instead of duplicating the MAC header, the Phonepl program inserts a constant MAC header that contains fixed source and destination MAC addresses and a TEB type value (*i.e.*, transparent Ethernet bridge, x6558).

As shown in **Figure 4**, the above program was used in an extended VNode, which is a gateway between slices and external networks and is called NACE or NC [17]. This network consists of the VNode and two personal computers, PC1 and PC2. PC1 simulates a terminal or a virtual node in a slice. PC2 is in an external physical network. The VNode connects the slice and the external network, and it must convert the packet format, *i.e.*, convert from the internal to external protocols, and vice versa, but the base component of the VNode does not have this conversion function. The VNode is experimentally extended by the node plug-in architecture with the

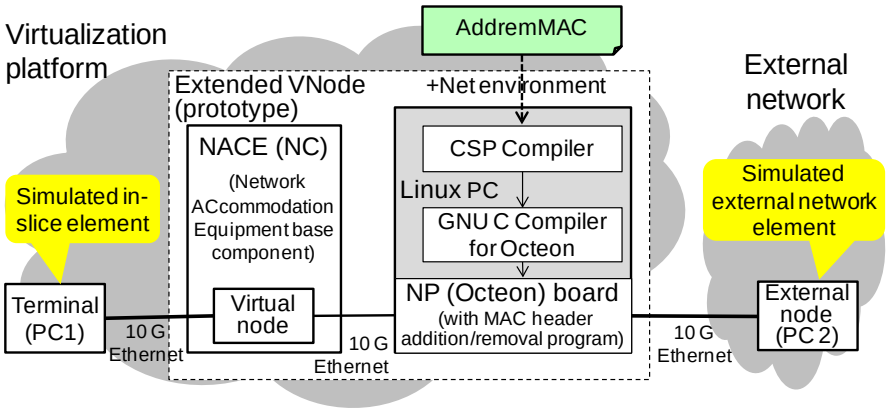


**Figure 4.** Extended VNode environment for experiments.

+Net environment, which consists of a PC with a Phonepl compiler, run-time routines, a GNU C compiler for Octeon, and WANic-56512 with twelve-core 750-MHz Octeon. By using conversion programs written in Phonepl, the VNode can adapt to various types of external networks.

Maximum performance of the test program was measured by using a network-measurement-tool suite called IXIA. Both operations, *i.e.*, MAC-header addition and deletion, were measured, and compared with a pass-through program, which is also written in Phonepl. The measurement results are shown in **Figure 5**. In this experiment, the input packet representation is mixed or cached, and the output packet representation is mixed or cached for header deletion and it is gathered for header addition. Uncached format is not used here because not whole cached data is removed by the header deletion. The maximum throughput (input rate) that can be passed with almost no packet drop is over 7.5 Gbps when the packet size is 256 bytes or larger. This throughput is close to the wire rate. The throughputs of two programs are mostly the same, indicating that the major overhead lies in the hardware or the initialization/finalization code, namely, not in the compiled code or the packet/string run-time routines.

**Table 1** compares the performance of the Phonepl program on the Octeon and a sequential C program on eight-core 3-GHz Intel Xeon processors. Although the performance of the former is much higher, it is mainly caused by the number of used cores. If all the cores are used, the throughput of Xeon may be better; however, it is very hard to use multiple cores and to preserve the order of packets in Xeon. As shown in **Table 1**, the Phonepl program is much shorter even when compared with the C program.

Moreover, **Table 1** suggests an important difference between the two implementations; that is, the packet loss ratio is slowly increasing in the Xeon implementation because cache miss is slowly increasing, but packets are almost never lost if the input ratio is 9.2 Gbps or less in the Phonepl implementation because the memory usage is completely controlled.

## 6.2. Timestamp Handler for Network Virtualization Platform

The second program, which is described in detail in another paper [18], is a program for measuring communication delay between two points in the virtualization network. In this evaluation, NPs and the program was used only in VNodes, and a slow-path program was used in the gateways.

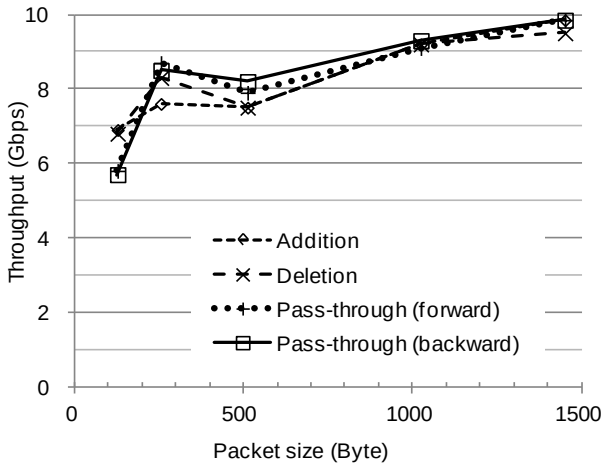


**Figure 5.** Performance of MAC-header addition/deletion.

**Table 1.** Results of MAC header addition/deletion.

| Implementation | Throughput (Gbps)* | | Program lines |
|---|---|---|---|
| | Header addition | Header deletion | |
| Phonepl program | 9.2† | 9.2† | 26‡ |
| C program (Xeon, single core)** | 2.3† (4.0††) | 1.7† (4.0††) | 161‡ |

*Packet size: 1024 B; **Promiscuous mode is used; †No packet loss (ratio < $10^{-6}$); ††Packet loss ratio = $10^{-3}$; ‡Comment-only lines are not counted.

A VNode platform can support delay measurement function without adding programs and data (*i.e.*, packet format) for measurement to programs in virtual nodes. This function is useful when slice developers want to measure delay of a high-bandwidth application with certain intelligent functions in relaying nodes. A special type of virtual links between nodes, which is called measurable VLAN virtual link (MVL) type and developed by using the VNode plug-in architecture, is used to implement this function. MVLs are implemented by using timestamp insertion/deletion programs in the nodes. A VNode removes the platform header, which includes a GRE/IP or VLAN header and the timestamp, from an incoming packet and adds one to an outgoing packet, so programs that handles packets on a slice never see the platform header.

The virtualization-network structure used for this experiment is drawn in **Figure 6**. Two terminals communicate using a slice. The physical network contains two VNodes. Each VNode contains a virtual node, which are connected by an MVL. In the platform, each packet has a platform header with a timestamp.

The communication and measurement methods used for this experiment is as follows. The timestamp is inserted at the entrance gateway. Each VNode generates a packet for the virtual node by removing the platform header from an incoming packet and restores the timestamp to outgoing packets that comes from the virtual node and are identified with a stored incoming packet. The timestamp is tested and deleted at the exit gateway, which calculates the delay between the entrance and exit gateways. In the network described in **Figure 6**, the two VNodes and one PC is used for the two gateways (and terminals) to avoid the difficult synchronization problem. Terminal PCs communicate each other by using Ethernet packets, which are switched by the MAC addresses in the virtual nodes. A WANic-56512 that contains the program handles both incoming and outgoing packets. An Ethernet switch program, which is a slow-path program, works on a virtual node in a VNode.

The NP also swaps the external and internal MAC addresses in the platform header [1]. To swap addresses, the program contains a conversion table for these MAC addresses, which is implemented using a string array, and accepts virtual-link-creation and deletion requests. A creation request adds an entry to the conversion table.

The results show the gateway-to-gateway delay is 178 μS ($\sigma$ = 24 μS). **Table 2** compares the performance of the 750 MHz Octeon and the 3-GHz Xeon. The performance is very close to wire rate. The C program is relatively short because this program does not contain conversion-table configuration code but the Phonepl program contains it. However, the former is still much longer.

## 7. Concluding Remarks

An open, portal, and high-level language, called Phonepl, is proposed. By using Phonepl, a programmer can develop a program that uses SRAM and DRAM appropriately without having to be aware of a distinction between SRAM and DRAM. To handle packets appropriately in this environment, four packet data-representations and packet-operation methods are proposed. A prototype using Octeon NP was developed and evaluated. The

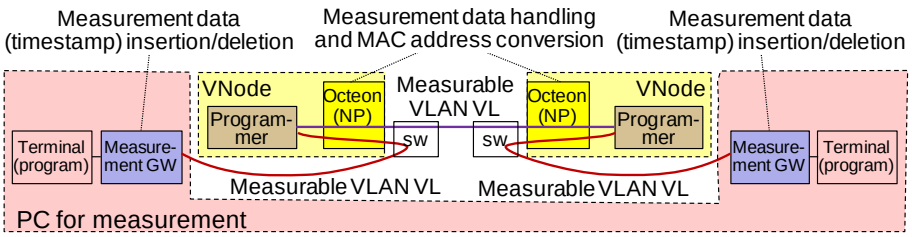


**Figure 6.** Virtualization-network structure for time-stamp handling.

**Table 2.** Results of timestamp handling and conversion.

| Implementation | Throughput (Gbps)* | | Program lines |
|---|---|---|---|
| | Header addition | Header deletion | |
| Phonepl program | 10.0† | 9.5† | 99‡ |
| C program (Xeon, single core)** | 2.3† (4.0††) | 2.2† (4.0††) | 190‡ |

*Packet size: 1024 B; **Promiscuous mode is used; †No packet loss (ratio < $10^{-6}$); ††Packet loss ratio = $10^{-3}$; ‡Comment-only lines are not counted.

throughput of the prototype system is close to the wire rate, *i.e.*, 10 Gbps, when the packet size is 256 bytes or larger, in several packet-conversion applications. Although this is a preliminary result, it proves the proposed method is promising in achieving our objectives, *i.e.*, popularity among developers, reduced cost in programmability, and portability.

Future work includes evaluation of Phonepl language and processor by human programmers and improvement of the language design and implementation according to the evaluation result. Although Phonepl and the language processor inevitably have limitations, they should be acceptable and, if possible, natural to programmers. Future work also includes implementation of Phonepl for other types of NPs to prove the portability. Moreover, the memory allocation and deallocation mechanism must be improved to reduce memory leak caused by global variable assignments and the performance of the Phonepl language processor should be improved.

## Acknowledgements

The author thanks Professor Aki Nakao from the University of Tokyo, Yasushi Kasugai, Kei Shiraishi, Takanori Ariyoshi, Takeshi Ishikura, and Toshiaki Tarui from Hitachi, Ltd., and other members of the project for discussions and evaluations. Part of the research results described in this paper is an outcome of the Advanced Network Virtualization Platform Project A funded by National Institute of Information and Communications Technology (NICT).

## References

[1] Kanada, Y., Shiraishi, K. and Nakao, A. (2012) Network-Virtualization Nodes That Support Mutually Independent Development and Evolution of Components. *IEEE International Conference on Communication Systems* (*ICCS* 2012). http://dx.doi.org/10.1109/iccs.2012.6406171

[2] Shah, N., Plishker, W., Ravindran, K. and Keutzer, K. (2004) NP-Click: A Productive Software Development Approach for Network Processors. *IEEE Micro*, **24**, 45-54. http://dx.doi.org/10.1109/mm.2004.53

[3] Fatahalian, K., Knight, T.J., Houston, M., Erez, M., Horn, D.R., Leem, L., Park, J.Y., Ren, M., Aiken, A., Dally, W. J. and Hanrahan, P. (2006) Sequoia: Programming the Memory Hierarchy. 2006 *ACM/IEEE Conference on Supercomputing*. http://dx.doi.org/10.1109/sc.2006.55

[4] Goglin, S.D., Hooper, D., Kumar, A. and Yavatkar, R. (2003) Advanced Software Framework, Tools, and Languages for the IXP Family. *Intel Technology Journal*, **7**, 64-76.

[5] Cavium Networks (2010) OCTEON Programmer's Guide. The Fundamentals. http://university.caviumnetworks.com/downloads/Mini_version_of_Prog_Guide_EDU_July_2010.pdf

[6] Bell, S., Edwards, B., Amann, J., Conlin, R., Joyce, K., Leung, V., MacKay, J., Reif, M., Bao, L.W., Brown, J., Mattina, M., Miao, C.-C., Ramey, C., Wentzlaff, D., Anderson, W., Berger, E., Fairbanks, N., Khan, D., Montenegro, F., Stickney, J. and Zook, J. (2008) TILE64-Processor: A 64-Core SoC with Mesh Interconnect. *IEEE International Solid-State Circuits Conference* (*ISSCC* 2008), San Francisco, 3-7 February 2008, 88-598. http://dx.doi.org/10.1109/isscc.2008.4523070

[7] Chen, M.K., Li, X.F., Lian, R., Lin, J.H., Liu, L.X., Liu, T. and Ju, R. (2005) Shangri-La: Achieving High Performance from Compiled Network Applications While Enabling Ease of Programming. 2005 *ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI*'05), 224-236. http://dx.doi.org/10.1145/1064978.1065038

[8] Kohler, E., Morris, R., Chen, B.J., Jannotti, J. and Frans Kaashoek, M. (2000) The Click Modular Router. *ACM Transactions on Computer Systems*, **18**, 263-297. http://dx.doi.org/10.1145/354871.354874

[9] Foster, N., Harrison, R., Freedman, M.J., Monsanto, C., Rexford, J., Story, A. and Walker, D. (2011) Frenetic: A Network Programming Language. 16*th ACM SIGPLAN International Conference on Functional Programming* (*ICFP*'11). http://dx.doi.org/10.1145/2034773.2034812

[10] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S. and Turner, J. (2008) Open Flow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, **38**, 69-74. http://dx.doi.org/10.1145/1355734.1355746

[11] Arasu, A., Babu, S. and Widom, J. (2006) The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, **15**, 121-142. http://dx.doi.org/10.1007/s00778-004-0147-z

[12] Monsanto, C., Foster, N., Harrison, R. and Walker, D. (2012) A Compiler and Run-Time System for Network Programming Languages. *Proceedings of the* 39*th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Philadelphia, 25-27 January 2012, 217-230. http://dx.doi.org/10.1145/2103656.2103685

[13] Nakao, A. (2012) VNode: A Deeply Programmable Network Testbed through Network Virtualization. *Proceedings of*

*the* 3*rd IEICE Technical Committee on Network Virtualization*, Tokyo, 2 March 2012. http://www.ieice.org/~nv/05-nv20120302-nakao.pdf

[14] Kanada, Y. (2013) A Node Plug-In Architecture for Evolving Network Virtualization Nodes. 2013 *Software Defined Networks for Future Networks and Services* (*SDN4FNS*). http://dx.doi.org/10.1109/sdn4fns.2013.6702531

[15] Kanada, Y. (2014) A Method for Evolving Networks by Introducing New Virtual Node/Link Types Using Node Plug-Ins. *Proceedings of the IEEE Network Operations and Management Symposium* (*NOMS*), Krakow, 5-9 May 2014, 1-8. http://dx.doi.org/10.1109/noms.2014.6838417

[16] Kanada, Y. (2014) Controlling Network Processors by Using Packet-Processing Cores. *Proceedings of the* 28*th International Conference on Advanced Information Networking and Applications Workshops* (*WAINA*), Victoria, 13-16 May 2014, 690-695. http://dx.doi.org/10.1109/waina.2014.112

[17] Kanada, Y., Shiraishi, K. and Nakao, A. (2012) High-Performance Network Accommodation into Slices and In-Slice Switching Using a Type of Virtualization Node. 2*nd International Conference on Advanced Communications and Computation* (*Infocomp* 2012), IARIA.

[18] Kanada, Y. (2014) Extending Network-Virtualization Platforms Using a Specialized Packet-Header and Node Plug-Ins. *Proceedings of the* 22*nd International Conference on Telecommunications and Computer Networks*, Split, 17-19 September 2014.

Scientific Research Publishing (SCIRP) is one of the largest Open Access journal publishers. It is currently publishing more than 200 open access, online, peer-reviewed journals covering a wide range of academic disciplines. SCIRP serves the worldwide academic communities and contributes to the progress and application of science with its publication.

**Other selected journals from SCIRP are listed as below. Submit your manuscript to us via either submit@scirp.org or Online Submission Portal.**