An Extensible Design of a Load-Aware Virtual Router Monitor in User Space

CHOI, Fu Wing

A Thesis Submitted in Partial Fulfilment

of the Requirements for the Degree of

Master of Philosophy

in

Computer Science and Engineering

The Chinese University of Hong Kong

September 2011


THE CHINESE UNIVERSITY OF HONG KONG
23 JAN 2013
UNIVERSITY
LIBRARY SYSTEM

Thesis/Assessment Committee

Professor XU, Qiang (Chair)

Professor LEE, Pak Ching (Thesis Supervisor)

Professor LUI, Chi Shing (Committee Member)

Professor WANG, Jilong (External Examiner)

Abstract of thesis entitled:

abstract

Router virtualization enables multiple virtual routers to be hosted on a physical shared substrate, and hence facilitates network management and experimentation. One critical issue of router virtualization is resource allocation of virtual routers. We explore this issue in the user-space design in order to allow extensibility and scalability.

In this thesis, we develop a user-space *load-aware virtual router monitor (LVRM)* atop a commodity multi-core architecture, with a key feature that it can dynamically manage CPU core resources among virtual routers based on their traffic loads so that they can fully exploit the parallelism of multi-core architectures. The main idea is that we can have various routing processes executing in separate cores simultaneously; and we have a LVRM centralize the process isolation and the resource monitoring (e.g., NIC). Also, LVRM adopts an extensible and scalable design so that each component can support different variants of implementation. We implement a proof-of-concept prototype for LVRM and empirically evaluate its performance overhead on top of a multi-core testbed. We show that LVRM can effectively manage the resources of different virtual routers based on their respective traffic loads. Our work provides insights into resource management in user space in the context of router virtualization. Source code of LVRM is available at

`http://ansrlab.cse.cuhk.edu.hk/software/lvrm`.

Submitted by CHOI, Fu Wing

for the degree of Master of Philosophy in Computer Science and Engineering

at the Chinese University of Hong Kong in July 2011

Abstract of thesis entitled:

摘要

路由器虛擬化允許多個虛擬路由器被寄存在一個實體共用的基板上，從而方便了網絡管理和實驗。一個路由器虛擬化的關鍵問題是虛擬路由器的資源分配。我們在用戶空間的設計中探討這個問題，以允許擴展性。

在這篇論文中，我們在一個商品的多核心架構上開發一個用戶空間的、*負載感知的和虛擬路由器的監控器*，具有一個重要功能：它可以動態地在虛擬路由器間根據其交通荷載而管理中央處理器的核心資源，使它們可以充分利用多核心架構的並行。主要的想法是，我們可以有不同的路由進程在不同的核心中同時執行；而我們有一個負載感知的和虛擬路由器的監控器集中進程隔離和資源監測（例如，網卡）。此外，負載感知的和虛擬路由器的監控器採用一個可擴展的設計，使每個組件都可以支持不同實施的變體。我們爲負載感知的和虛擬路由器的監控器而實施一個概念驗證的原型，及經驗地在多核心試驗台上評估其效能開銷。我們表明，負載感知的和虛擬路由器的監控器可以有效地根據各自的交通負荷管理不同虛擬路由器的資源。在路由器虛擬化的背景中，我們的工作提供在資源管理中的和在用戶空間中的洞察。負載感知的和虛擬路由器的監控器之源代碼在

http://ansrlab.cse.cuhk.edu.hk/software/lvrm 。

Submitted by CHOI, Fu Wing

for the degree of Master of Philosophy in Computer Science and Engineering

at the Chinese University of Hong Kong in July 2011

Contents

List of Figures

Declaration

The Thesis submitted is partially based on our original work:

- Harry F. W. Choi and Patrick P. C. Lee

  "An Extensible Design of a Load-Aware Virtual Router Monitor in User Space"

  7th International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS) (in conjunction with ICPP'11), Taipei, Taiwan, September 2011.

Keywords

Router virtualization, resource allocation, multi-core

Chapter 1

Introduction

The virtualization technology simplifies process management by having multiple software instances hosted on a shared hardware substrate, and evolves as a solution to reduce hardware footprints. Specifically, in the context of packet forwarding and routing in networks, *router virtualization* enables multiple virtual routers to be hosted on shared network resources, such that each virtual router has its own data forwarding plane and is independently configured with its own set of routing policies. Thus, a virtual router works like a typical physical router. There have been commercial vendors (e.g., [9, 22]) that develop router products with router virtualization, in which a single physical router provides a platform for hosting multiple virtual (logical) routers. Therefore, we believe that router virtualization will be adopted in various practical applications. One example is to deploy a single physical router on a campus backbone network that provides connectivity for the IP subnets of different departments [14]. Each department can be assigned a set of virtual routers (hosted inside the physical router) and it can individually configure its own routing policies on each virtual router. A more recent application of router virtualization is network experimentation (e.g., VINI [3], OpenFlow [26]), where

users can form a network of virtual routers (or switches) and conduct controlled wide-area network experiments atop a shared network platform.

Instead of hosting virtual routers on physical routers, an alternative of deploying router virtualization is to host software-based virtual routers atop commodity, general-purpose hardware and operating systems, so as to trade processing speed for extensibility and programmability. Software routers (e.g., Click [21] and XORP [19]) emulate the routing functionalities of hardware routers, and allow flexible extensions and re-engineering of such functionalities. Given the emergence of multi-core technologies and advances in hardware architectures, it is shown that software-based router virtualization can be feasibly deployed using commodity hardware [14], such that the aggregate performance of software virtual routers is close to that of a single software router without virtualization.

To exploit the full potential of software-based router virtualization, a critical design issue is the *resource management* of virtual routers. Specifically, virtual routers may receive different amounts of data traffic load for their respective networks, and require different shares of resources (e.g., I/O, CPU, memory) for processing such packets in a fair manner. One approach is to rely on a general-purpose *hypervisor* (also called *virtual machine monitor*), such as Xen [2], for resource management by running each virtual router inside a virtual machine [16]. However, such an approach typically involves unnecessary overhead of processing operating system tasks besides routing functions. Also, it is unclear whether such a general-purpose hypervisor effectively adapts toward different network traffic patterns that are specific for router virtualization. Thus, it is desirable to have a customized, lightweight hypervisor that is capable of performing effective resource management specifically for router virtualization.

In this thesis, we propose a user-space *load-aware virtual router monitor (LVRM)*

that seeks to achieve resource management of virtual routers based on their data traffic loads. LVRM can in essence host different implementations of virtual routers, as long as we allow minimal changes to the interfaces of the virtual routers to enable them to interact with LVRM. Specifically, we focus on the deployment of software-based virtual routers atop a commodity multi-core architecture, and we narrow down our focus into one issue: *how to dynamically assign CPU cores to different virtual routers based on their data traffic loads?* LVRM addresses this question by considering different design dimensions, including: (i) core allocation, (ii) load balancing, (iii) load estimation, and (iv) inter-process communication. For each design dimension, LVRM allows extensibility for different variants of implementation, so as to adapt to different application requirements.

Through the extensible design of LVRM, our goal is to explore a set of design guidelines of resource management in router virtualization. We propose an extensible design of LVRM, and implement a proof-of-concept prototype of LVRM atop a multi-core architecture. Using extensive empirical experiments, we demonstrate that LVRM incurs minimal performance overhead in data forwarding in terms of throughput and latency when compared to native Linux IP forwarding. In addition, LVRM can support dynamic core allocation and load balancing of virtual routers based on their traffic loads. Our experimental results justify the feasibility of resource management in user space in the context of software-based router virtualization atop commodity multi-core architectures.

The remaining of the thesis proceeds as follows. Chapter 2 overviews the design of LVRM, and Chapter 3 elaborates the design details. Chapter 4 presents the experimental results for LVRM running atop a multi-core platform. Chapter 5 reviews related work, and Chapter 6 concludes.

Chapter 2

Overview

In this chapter, we overview the router virtualization architecture that we consider. We mainly address dynamic resource allocation in router virtualization. Specifically, we focus on the allocation of CPU processing resources among virtual routers (VRs) atop a commodity multi-core architecture. We present the design of a *load-aware virtual router monitor (LVRM)*, in which one key feature is to achieve dynamic allocation of CPU cores for VRs based on their traffic loads, so that each VR receives fair allocation of CPU processing power to process packets. Also, LVRM adopts an extensible design in its components. We first overview the router virtualization architecture that we consider, and then describe the major components of LVRM that collaboratively achieve the goal of resource allocation among VRs. Note that the implementation that we consider in this thesis is based on C++ and is running atop Linux.

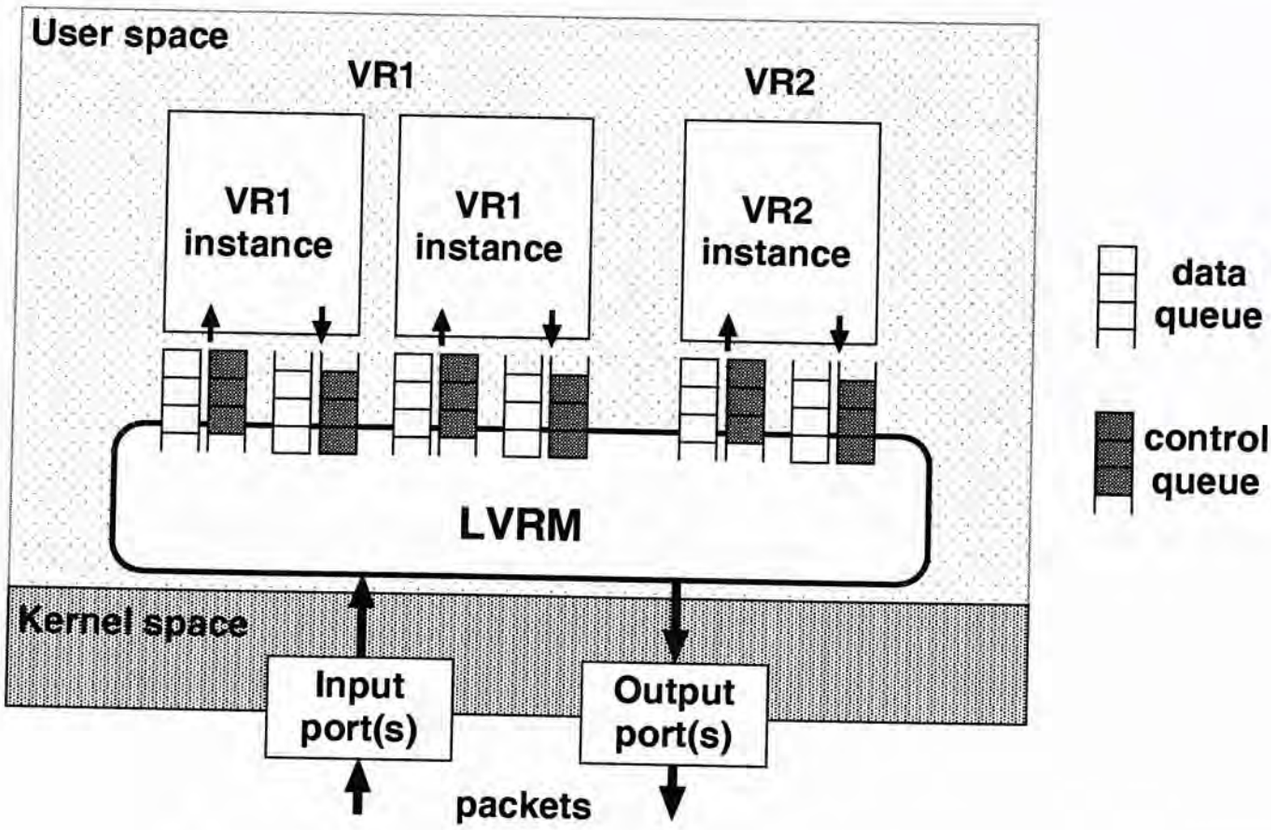

Figure 2.1: Overview of the router virtualization architecture.

2.1 Summary of our Router Virtualization Architecture

Our main goal is to virtualize the data forwarding planes of multiple virtual routers atop a shared hardware substrate. In our design, LVRM is a centralized process that manages a number of VRs, each of which is an independently administered router and has its own set of routing policies and configurations. Depending on the current traffic load, LVRM spawns one or multiple *VR instances (VRIs)* for each VR to process packets. The VRIs that belong to the same VR are expected to share the same set of routing policies and configurations. Figure 2.1 depicts a high-level overview of the entire router virtualization architecture that consists of LVRM and the VRIs created for different VRs.

We run both LVRM and VRIs as *user-space software-based processes* that can be

deployed on commodity, general-purpose multi-core architectures and operating systems. Running the processes in user space enables better programmability and extensibility, with a trade-off of degraded data forwarding performance as compared to the kernel space. It has been shown that software routers running in user space have slower data forwarding performance than in kernel space [25]. On the other hand, if we leverage concurrent lock-free synchronization of inter-process communication (IPC) [23] and kernel modules of packet capture acceleration [12] (see Chapter 3), then our experiments show that we can improve the data forwarding throughput performance (see Chapter 4 for details).

To understand the workflow of our router virtualization architecture in Figure 2.1, we present the forwarding path of a data frame from input to output. Suppose that each hosted VR is configured with an IP subnet and is responsible for processing data packets originated from this subnet, and that it is configured with the mappings of the routes to the network interfaces of the deployment architecture. The workflow is summarized as follows:

1. First, LVRM captures a raw data frame (in the Ethernet layer) from an input network interface.

2. LVRM inspects the source IP address of the data frame, and determines the VR that will process the data frame. It then dispatches the data frame to a VRI of the VR via an IPC queue called the *data queue*. Each VRI is associated with a pair of incoming/outgoing data queues. The dispatch decision of which VRI will process the data frame is based on the number of VRIs that have been spawned and the currently used load balancing scheme.

3. The data frame is then processed by the corresponding VRI. If the VRI forwards the data frame, then it indicates the output network interface in the data frame.

4. The VRI relays the data frame to its associated outgoing data queue. LVRM then sends the data frame to the correct output network interface.

Also, as shown in Figure 2.1, a VRI can share control information with other VRIs of the same VR, for example, to synchronize the routing state. The sharing is performed by associating each VRI with another pair of incoming/outgoing queues called the *control queues*. We assume that a control queue has a higher priority than a data queue. Thus, each VRI first processes any control event available in its incoming control queue, and then processes data frames available in its incoming data queue.

Chapter 3

LVRM Design

In this chapter, we describe the major components of LVRM that collaboratively achieve the goal of resource allocation among VRs. Note that the implementation that we consider in this thesis is based on C++ and is running atop Linux.

Inside LVRM, its design is built on several major user-space components arranged in a hierarchical structure. Figure 3.1 shows the internal design of LVRM, which can be viewed as a hierarchical structure. The hierarchical design of LVRM enables it to host multiple VRs, and each VR can host multiple VRIs. In this section, we explain in detail the features of each component, and justify how each component provides extensibility for different variants of implementation.

3.1 Socket Adapter

The socket adapter is the software interface that relays data frames via LVRM. LVRM can obtain a data frame by contacting the socket adapter, which then polls for available data frames from a lower-level interface (e.g., the kernel or the NIC). From the point of view of LVRM, the polling process of the socket adapter is transparent. The socket

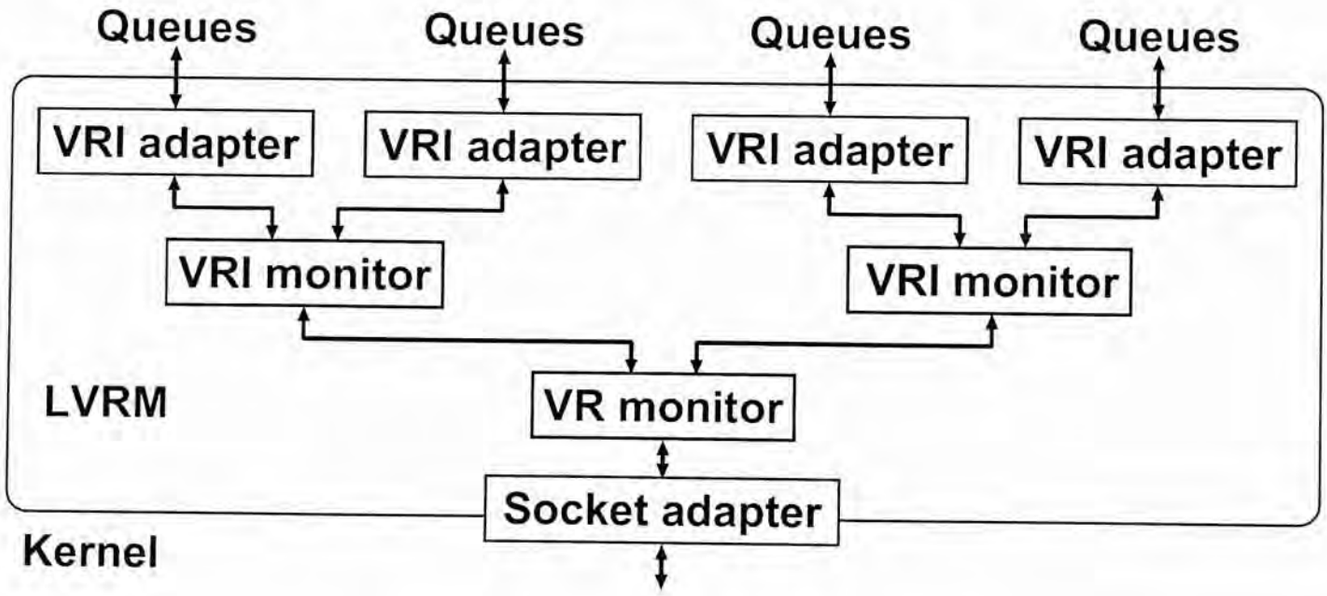

Figure 3.1: Hierarchical design inside LVRM.

adapter is also responsible for forwarding any data frames from LVRM to the lower level.

Currently, the socket adapter supports three variants of implementation of accessing data frames in the lower level:

Raw socket [28]. It is the interface between user-space applications and the kernel network stack for sending/receiving raw frames over the network. Our implementation is based on the BSD socket, with which we create a socket descriptor to access raw frames that start at the link layer (e.g., the Ethernet layer). We use the system call `recvfrom()` to retrieve raw frames via non-blocking polling, and use the system call `send()` to send raw frames.

PF_RING [12]. It is a new socket type that is designed for speeding up data capture in network monitoring. Its idea is to poll the NIC directly and retrieve raw frames from the NIC through the zero-copy technique, in order to save the unnecessary kernel memory allocation/deallocation as in the raw socket case. Note that, before PF_RING version 3.7.5 (February 2011), it only considers how to retrieve incoming frames, but does not consider how to send outgoing frames. Thus, in LVRM version 1.0 (23 February 2011)

the socket adapter still sends outgoing frames via the raw socket. Currently, in LVRM version 1.1 (3 September 2011), it both considers how to retrieve incoming frames and how to send outgoing frames. Thus, the socket adapter can send outgoing frames via either the raw socket or the PF_RING new function call `ipfring_send()`.

Main memory. We also enable the socket adapter to receive raw frames from main memory rather than from the network. The idea is to exclude the performance bottleneck in the network, so that we can evaluate the processing overhead mainly due to LVRM. We load a trace of raw frames into main memory, from which the socket adapter can sequentially retrieve the raw frames.

3.2 VR Monitor

LVRM is by itself a user-space process, and it internally has a major component *VR monitor* that coordinates different VRs. In particular, it is responsible for *core allocation*, which coordinates how different VRs use CPU core resources within the underlying multi-core architecture. It adjusts the number of cores being allocated for each VR based on its traffic load. To avoid the contention of multiple processes for a single CPU core, it is important to associate a CPU core with only one VRI.

Here, we consider two core allocation approaches as summarized below:

Fixed approach: The VR monitor pre-assigns a fixed set of cores to a VR when the VR first starts.

Dynamic approaches: Figure 3.2 summarizes the algorithm of the dynamic approach. The VR monitor re-assigns a dynamic set of cores to a VR when LVRM receives a packet after a second or more from the previous re-assignment. In particular, we consider two variants of the dynamic approach:

```
/* Create VRI adapter:  adding a new VR */
/* instance for a VR */
```
function VRI monitor's "create VRI adapter"(int `CPU_ID`)
1: Create packet queues and event queues
2: Put shared queues into shared memory
3: Bind VRI to run on the core specified by `CPU_ID`
4: Add VRI to the list of VRI

```
/* Destroy VRI adapter:  deleting a VR */
/* instance for a VR */
```
function VRI monitor's "destroy VRI adapter"(int `CPU_ID`)
1: Kill the VRI running on the core specified by `CPU_ID`
2: Destroy all queues and clear allocated memory
3: Remove VRI from the list of VRI

```
/* Allocate:  called upon receipt of a */
/* packet after 1s or more from previous */
/* core allocation/deallocation process */
```
function Core allocator's "allocate"()
1: **for** each VR **do**
2: **if** arrival rate $\ll$ threshold(service rate w/ 1 less VRIs) **then**
3: **return** This VRI monitor destroys VRI adapter(best CPU)
4: **else if** threshold(service rate) $\ll$ arrival rate **then**
5: **return** This VRI monitor creates VRI adapter(best CPU)
6: **end if**
7: **end for**

Figure 3.2: Algorithm of dynamic approach.

- *Dynamic approach with fixed thresholds.* The VR monitor assigns cores to a VR based on the traffic load of the VR. If the current traffic load of the VR is above a threshold, then the VR monitor allocates an additional CPU core to the VR; if the traffic load of the VR is low, then the VR monitor deallocates a CPU core from the VR. Strength of this dynamic approach is the simplicity, as it uses a rule based on only the packet rate for the allocation. The possible weakness may be the reliability if some VRs may require much more processing power (for instance, because of much more rules). Currently, we measure the load of a VR by estimating the exponential weighted average arrival rate of incoming data frames for the VR.

- *Dynamic approach with dynamic thresholds.* The VR monitor assigns cores to a VR based on the traffic load and also the service rate of the VR. If the current traffic

load of the VR is above the current service rate, then the VR monitor allocates an additional CPU core to the VR; it means that the VR needs more cores as the load is over the service capacity of VRIs. If the traffic load of VR is lower than the service rate with one less VRIs of VR, then VR monitor deallocates a CPU core from VR; similarly, it means that VR is capable to serve traffic load with less VRIs. One of the strengths of this dynamic approach is that it is more straightly for indicating the needs of the VRIs. Currently, we measure the load of a VR by estimating the exponential weighted average arrival rate of incoming frames for the VR. Also, currently we measure the service rate of a VR by estimating the average departure rate of the incoming data queues for the VR. One of the strengths of this departure rate rather than the CPU load via function call `getrusage()` is that it can be compared with the traffic load more directly in order to indicate the needs of the VRIs.

We expect that the dynamic approach is more resource-efficient than the fixed approach, since it allocates cores based on the traffic load and hence avoids over-provisioning. We also consider two special heuristics to improve the performance of the dynamic approach. First, LVRM is a user-space process that we bind to a CPU core. It is intuitive to first assign a VR the cores that are "close" to LVRM, so as to minimize inter-core communication between LVRM and the VR. Thus, the dynamic approach first allocates the *sibling cores*, i.e., the cores that reside in the same CPU as the core on which LVRM is running, followed by the *non-sibling cores* (i.e., cores in a different CPU). We examine the impact of affinity in core allocation in Chapter 4.

Second, it is important to control how often the core allocation/deallocation process should take place. If the frequency is too high, then it will cause instability to the

performance of the VR; if the frequency is too low, then it will result in poor responsiveness to the load conditions. Thus, the dynamic approach periodically monitors the traffic load of each VR, and triggers the core allocation/deallocation process if necessary. Here, we set the period to be 1 second, while this parameter is tunable depending on the applications. In general, our experiments show that the core allocation/deallocation process has a small reaction time (see Chapter 4).

The design allows flexible changes, for example, to extend via the function call `setrlimit()` with other resource managements such as the memory management. We consider CPU more, as the loads are usually CPU-intensive; routers use the memory usually for the summarized routes, which are less intensive to the commodity hardware. The capacity of the memory is seldom a major concern.

3.3 VRI Monitor

A VRI monitor is associated with each VR, and is to coordinate the VRIs of a VR. It creates or deletes VRIs via the function calls `vfork()` and `kill()`, respectively, based on the number of cores assigned by the VR monitor (assuming that one core is for only one VRI). It is also responsible for *load balancing*, which balances the CPU core resources among the VRIs of the same VR. Figure 3.3 summarizes the algorithms of the load balancing. Specifically, it dispatches frames to different VRIs for processing, so that the VRIs receive balanced shares of processing loads. Here, we consider three implementations of load balancing:

Join-the-shortest-queue. It forwards data frames to the VRI that currently has the lightest traffic load, where the load is estimated based on the load estimation algorithm (see the description of the VRI adapter below).

```
/* JSQ: join-the-shortest-queue */
function Load balancer's "JSQ"()
 1: for each VRI in this VR do
 2:    if queue load of this VRI < load of current shortest queue then
 3:       Remember the VRI with the current shortest queue load
 4:    end if
 5: end for
 6: return the VRI with the current shortest queue load

/* Rnd/RR: random/round-robin */
function Load balancer's "Rnd"()/"RR"()
 1: return the randomly-selected/next and valid VRI

/* Balance:  called upon receipt of a */
/* packet */
function Load balancer's "balance"(char* buffer)
 1: if the implementation is defined as flow-based then
 2:    Locate the TCP/IP headers from the buffer
 3:    Construct the flow entry from the headers
 4:    Hash table find the entry with current timestamp and add flag
 5:    if the entry is found and the VRI of the entry is valid then
 6:       return the VRI of the entry
 7:    end if
 8: end if
 9: return (if flow-based, VRI of added entry ←) JSQ()/Rnd()/RR()
```

Figure 3.3: Algorithms of load balancing.

Random. It forwards each data frame to a VRI that is uniformly selected among all available VRIs.

Round-robin. It forwards packets to each VRI in a round-robin manner.

Note that the above implementations can be *flow-based* or *frame-based*, in which we dispatch data frames to VRIs on a per-frame basis. Another type of implementation is *flow-based* (e.g., see [13]), in which data frames of the same flow (e.g., based on 5-tuples) are always forwarded to same core. Our flow-based load balancing is similar to the frame-based load balancing. Instead, we dispatch the second or later data frames to the VRIs based on the first data frame of the same flow. Instead of the dynamic arrays, the hash tables are used for the performance issues in the connection tracking functions, which are called for each incoming data frames. The flow-based implementation avoids reordering

of data frames that belong to the same flow. Note that the VRI monitor can support both frame-based and flow-based load balancing without affecting the design of other components. We examine the impact of load balancing in Section 4.4 and 4.5.

3.4 VRI Adapter

A VRI adapter is associated with each VRI, and is to relay data packets to/from the VRI. It is also responsible for *load estimation* of the VRI, and reports the estimated load values to the VRI monitor for load balancing. Figure 3.4 shows the algorithms of the load estimation. While there are many variants of load estimation, we consider a simple version as follows. When the VRI adapter forwards a data frame to the VRI, it measures the load by observing the current queue length. It then computes the exponential weighted average queue length of the incoming data queue of each VRI.

3.5 Inter-Process Communication (IPC) Queue

An IPC queue enables two processes (i.e., the *producer* and the *consumer*) to share information, such that the producer (consumer) process inserts (extracts) items to (from) the queue in a first-in-first-out manner. Each VRI is associated with two types of IPC queues: (i) data queues and (ii) control queues (see Figure 2.1). Each VR can send/receive data frames to/from its VRIs via a pair of incoming/outgoing data queues, while each pair of VRIs can exchange control events via a pair of incoming/outgoing control queues.

It is important to minimize the inter-process communication. Thus, we consider an IPC queue implementation based on *lock-free synchronization* [23]. It allows the producer and consumer processes to simultaneously access the queue, so long as they do not access

```
class Load balancer
 1: double Average_Load

/* Get estimate:  called upon load */
/* balancing */
function Load balancer's "get estimate"()
 1: return the Average_Load

/* Arrival time */
function Load balancer's "arrival time"()
 1: if the current time stamp is valid then
 2:     The current time stamp becomes the previous time stamp
 3:     Get the new time stamp for the current time stamp
 4:     Update the current load to be: current time stamp - previous timestamp
 5: end if

/* Queue length */
function Load balancer's "queue length"()
 1: Update the current load to be the VRI adapter's ring buffer's data count

/* Estimate:  called upon receipt of a */
/* packet */
function Load estimator's "estimate"()
 1: Get the "arrival time"()/"queue length"() for the current load
 2: if the Average_Load is valid then
 3:     average_Load ← (current load + weight × Average_Load) / (1 + weight)
 4: end if
```

Figure 3.4: Algorithms of load estimation.

the same queue entry. It is more efficient than the lock-based synchronization, in which only one process can access the queue at one time. Our current lock-free queue implementation is based on [23], while other improved lock-free queue implementations [17, 24] can also be used in LVRM.

3.6 LVRM Adapter for VRI

Similar to the VRI adapter, a LVRM adapter is for each VRI to associate with LVRM, and is to relay data packets to/from the LVRM. Instead of accessing the IPC queues directly, we provide the Application Programming Interface (API) that allow the VRIs to simply communicate with LVRM via the function calls `fromLVRM()` and `toLVRM()`.

The LVRM adapter is initialized with a shared memory identifier, which is passed from LVRM via the main arguments to VRIs. If the dynamic thresholds of the dynamic core allocations are enabled, the LVRM adapters are also responsible for estimating the service rates of the VRIs, and report the estimated values of the service rates via the IPC queues to the LVRM. Here, we consider a simplified version as follows. When the LVRM adapter forwards a data frame from the IPC queue to the VRI, it measures the service rate by observing the service time between the current call and the next call of the function `fromLVRM()`. It then computes the average service rate of the incoming data frame of each VRI.

3.7 VRI

A VRI is for its corresponding user to manipulate the data and the control messages, and is also responsible for interpreting the address resolution and routing information. Currently, the route tables are initialized with the map files, which pass the static routes to the memories of the VRIs. If dynamic routes are used, the VRIs can be slightly changed to support both static and dynamic routes without affecting the design of LVRM. Instead of fixing an application protocol for inter-VRI communication, we allow users to communicate with each other VRIs via their user-specified protocols similar to the UDP socket programming. When the VRI sends/receives a control message to/from others, they access the control messages in the ways similar to accessing datagrams. Here, we consider a simplified version of VRIs, which do the minimal routing. The packet processing of the VRIs is not our major concern.

3.8 Interfacing Between LVRM and VRs

LVRM is designed with the capability of hosting different implementations of VRs, provided that we allow minimal changes to the interfaces of the VRs so that the VRs can communicate with LVRM. Specifically, instead of accessing data frames via network interfaces, the VRIs of each VR should now access data via the IPC queues. LVRM allocates a shared memory segment for each IPC queue (via the function call `shmget()`). The shared memory segment is associated with a shared memory identifier, through which LVRM and VRIs can access.

Each VR implements the essential data forwarding/routing functions as a software-based router. It can spawn multiple VRIs for processing raw frames. Note that the internal processing of the VRI on the raw frames is transparent to LVRM.

We consider two types of user-level VRs to be hosted by LVRM, including (i) *C++ VR*, a simple data forwarding program written in C++ and (ii) *Click VR*, a forwarding program based on Click Modular Router [21]. By default, both types of VRs perform the minimal data forwarding function, i.e., by simply relaying data frames from an input network interface to an output network interface. Note that the Click VR parses a configuration script to conduct the forwarding function, and internally relays data frames via different modules. Thus, we expect that the C++ VR is more lightweight and can eliminate the internal processing overhead in Click.

Chapter 4

Experiments

In this chapter, we conduct empirical studies on LVRM and evaluate its performance overhead. The goals of our empirical studies are three-fold. First, we show that LVRM incurs minimal performance overhead, even it is deployed in user space. Second, we show that LVRM is load-aware, in the sense that it dynamically allocates core resources for VRIs with regard to the current loads of forwarding traffic. Third, we show that LVRM is scalable, even in other complicated cases.

4.1 Experimental Setup

Testbed. Figure 4.1 shows the testbed where we conduct our experiments. Similar to those in [16, 21, 25], the testbed is composed of two sub-networks that connected by a gateway, on which we deploy LVRM. The sub-networks and the gateway are connected via 1-Gigabit switches and network interfaces (i.e., the raw network bandwidth is 1Gbps, which is widely in use). We put two sender hosts (S_1 and S_2) on one sub-network, and two receiver hosts (R_1 and R_2) on another sub-network. We have senders S_1 and S_2 generate raw frames (in layer 2) to receivers R_1 and R_2 via the gateway, respectively.

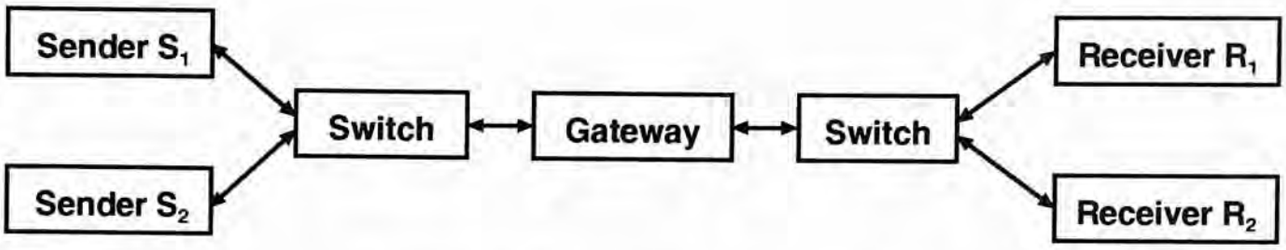

Figure 4.1: The experimental topology.

The gateway is deployed on a machine with two Intel Xeon E5530 64-bit quad-core 2.4GHz CPUs (i.e., a total of eight cores) and 8GB RAM. The sender and receiver hosts are deployed on machines with two Intel Xeon 64-bit dual-core W3565 3.2GHz CPUs and 2GB RAM. All machines are running Linux 2.6.35 with Ubuntu 10.10. The implementation is based on C++, and is compiled with GCC 4.4.5 with the `-O3` option.

Before we conduct our experiments, we first evaluate whether our testbed can reflect a realistic network environment. In particular, we consider the *maximum frame rate* [7] achievable by the gateway in forwarding data traffic. To obtain our measurements, we enable Linux IP forwarding in the gateway, so that it can relay traffic from the senders to the receivers. Each sender host generates raw frames to its respective receiver host using the *minimum frame size* of an Ethernet frame [7], which is 84 bytes (including the preamble, payload, and check sequence). We obtain the maximum frame rate by increasing the sending rate of each sender host until the sending rate and the receiving rate differ by more than 2%. Based on our measurements, we find that both sender hosts can simultaneously send at most 224K frames per second (fps) based on our requirement. Thus, the maximum frame rate achievable by the gateway is 2×224 Kfps = 448 Kfps. This value lies in the range of the maximum frame rates achievable by commercial routers (e.g., 225 Kfps for a Cisco 3745 router [8] and 2 Mfps for a Cisco 7200 router [10]). Thus, we believe that our testbed can realistically resemble a routing network.

In our experiments, we have LVRM host two types of VRs: C++ VR and Click VR

(see Chapter 3). Both VRs perform the minimal data forwarding function by relaying raw frames from the interface of the sender sub-network to the interface of the receiver sub-network, as shown in Figure 4.1.

Traffic Model. We are interested in two traffic models:

- *The UDP/IP senders and receivers.* We have a coordinator generate the "START" requests to the senders S_1 and S_2 via a switch at the same moment. We have the senders start generating UDP/IP packets after receiving the "START" requests, respectively. For each sender, it generates UDP/IP packets once it finds that the aggregate source rate is lower than the specified source rate. The source models are constant departure. The resulted UDP flows are smooth and evenly distributed to the senders and the receivers. Unless otherwise specified, the flows last for 60 seconds and there are ten trials in one experiment.

- *Realistic FTP/TCP servers and clients.* Instead of the UDP/IP senders and receivers, we use the pre-installed FTP client programs `ftp` and install the FTP server programs `ftpserver` from the default updater. We have the FTP clients log-in anonymously via the gateway at the same moment. They generate bi-directional TCP/IP packets when they are getting some large files. The pair of FTP flows includes the data and the control connections in various flow and segment sizes. The aggregate source rates should be slightly lower than the maximum source rate controlled by the mechanisms of TCP. The resulted TCP flows are not as smooth as the UDP flows as mentioned above, but are also evenly distributed to the hosts. Unless otherwise specified, one trial of an experiment lasts for 600 seconds and there are three trials in one experiment.

Default implementation of LVRM. Unless otherwise specified, we assume the fol-

lowing default implementation of LVRM. We assume that the socket adapter is based on PF_RING for both retrieving incoming frames and sending outgoing frames. LVRM uses dynamic core allocation with fixed thresholds, and uses the frame-based join-the-shortest-queue scheme for load balancing.

Metrics. We are interested in three metrics:

- *Achievable throughput.* It corresponds to the maximum frame rate achievable by LVRM such that the sending rate and the receiving rate differ by no more than 2%.
- *Round-trip latency.* It corresponds to the average round-trip time obtained via the ICMP Ping utility. We generate 400K ICMP echo requests from a sender host to a receiver host, and measure the average round-trip time for the sender host to obtain the ICMP echo replies from the receiver host.
- *Fairness.* We are interested in two fairness indexes: (i) the *Jain's fairness index* [20], which focuses on the majority of the flows, and (ii) the *max-min fairness*, which focuses on the outliner.

4.2 Performance Overhead of LVRM

We first evaluate the performance overhead of the data path in LVRM. We seek to address the following questions:

- Given that LVRM is deployed in user space, does it incur significant performance overhead in data forwarding?
- Given that LVRM targets only data forwarding, is it more lightweight than general-purpose hypervisors that are designed for monitoring virtual machines?

In this section, we consider the case where LVRM hosts a single VR, and the VR uses only a single VRI to process raw frames. In Sections 4.3, 4.4 and 4.5, we consider how LVRM hosts a single VR with multiple VRIs, how LVRM hosts multiple VRs and how scalable LVRM is.

Experiment 1a (Achievable throughput in data forwarding). In this experiment, we aim to show that LVRM will *not* become a performance bottleneck in data forwarding throughput. Using the topology in Figure 4.1, we have both sender hosts generate raw frames of different frame sizes via the gateway to their respective receiver hosts, and then measure the achievable throughput. Here, we consider three types of data forwarding mechanisms deployed in the gateway:

- *Native Linux IP forwarding:* We enable the IP forwarding function in the gateway, and the forwarding decision is made within the Linux kernel.

- *LVRM:* We disable Linux IP forwarding, and have LVRM forward raw frames. Specifically, upon receiving raw frames from the input network interface of the sender sub-network, LVRM relays the raw frames to the VR that is being hosted, and the VR relays the raw frames to the outgoing network interface of the receiver sub-network. In particular, we consider three variants of LVRM:

  - *LVRM with C++ VR and raw socket*, in which LVRM hosts a C++ VR and uses non-blocking polls of the system call `recvfrom()` to retrieve raw frames from the network interface,

  - *LVRM with C++ VR and PF_RING*, in which LVRM hosts a C++ VR and uses the PF_RING library [12] to retrieve raw frames, and

– *LVRM with Click VR and PF_RING*, in which LVRM hosts a Click VR and uses PF_RING.

We assume that each VR uses a single VRI for forwarding raw frames. In later experiments, we also study how multiple VRIs further improve the forwarding performance.

- *General-purpose hypervisors:* We consider two publicly known general-purpose hypervisors VMware Server [30] and QEMU-KVM [4]. In each of the hypervisors, we host a guest virtual machine (VM), on which we install Linux and enable the IP forwarding function. We set the network adapter of each guest VM to bridged mode, so as to allow the guest VM to forward data frames. Each of the hypervisors relays traffic to the guest VM, which then relays traffic to the receiver sub-network through its hypervisor.

Figure 4.2 shows the achievable throughput of different data forwarding mechanisms versus the frame size[1]. First, we observe that the native Linux IP forwarding has the highest achievable throughput for all frame sizes. This result is expected, since the data path is the simplest among all the mechanisms.

The throughput performance of the general-purpose hypervisors (i.e., VMware Server and QEMU-KVM) is significantly worse than the native Linux IP forwarding. The reason is that in addition to data forwarding, they also incur performance overhead of processing various operating system tasks. We observe that QEMU-KVM has significantly poor performance. We do not know the exact reason, but we conjecture that the performance may be improved with other configuration settings.

[1]It is expected that for small frames, the throughput is less than the raw bandwidth 1Gbps, mainly due to the processing overhead of a large number of frames.

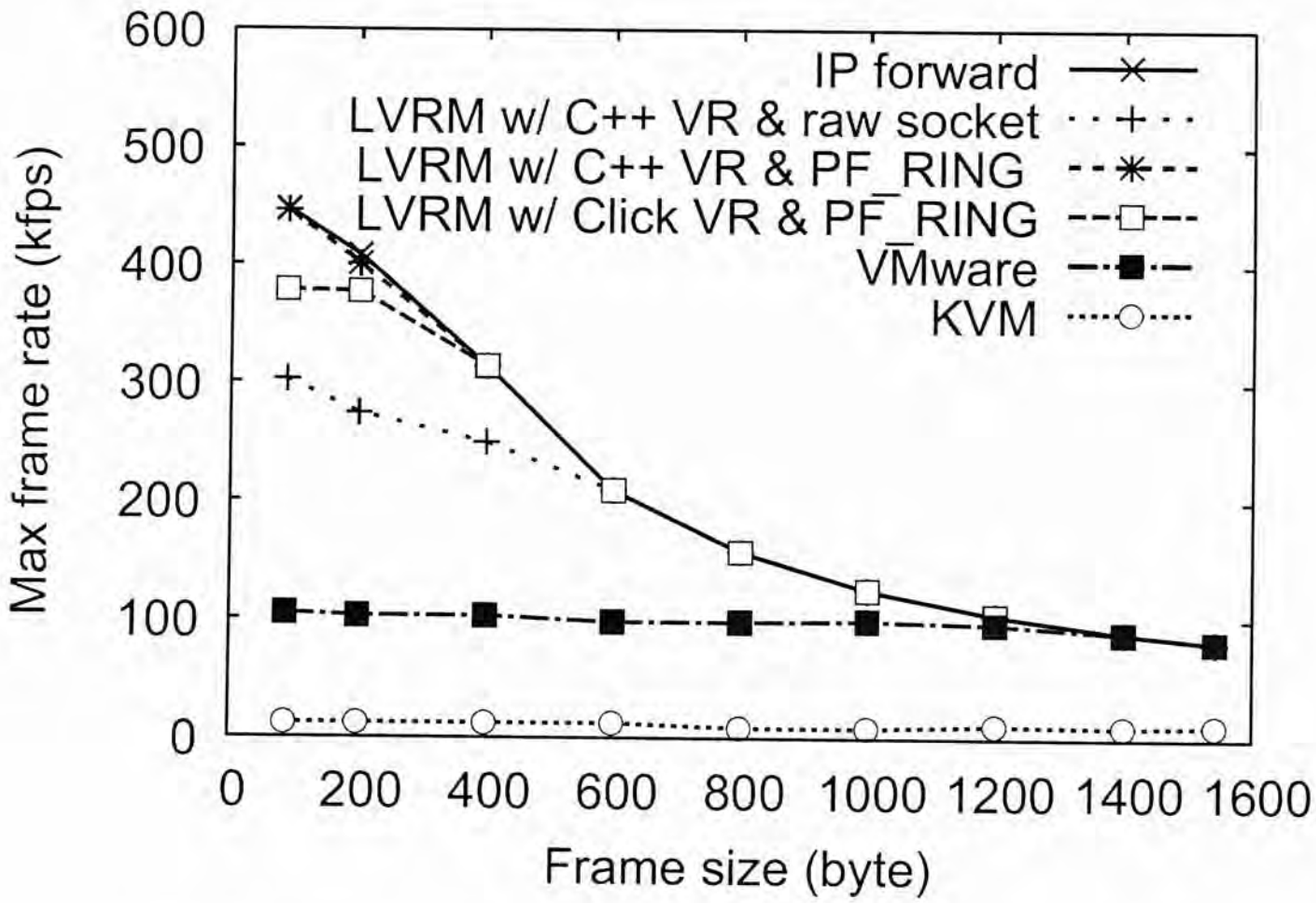

Figure 4.2: Experiment 1a: Achievable throughput in data forwarding.

For LVRM, it generally achieves higher throughput than the general-purpose hypervisors. We note that using the Click VR has smaller throughput than the C++ VR, since the Click VR has more internal operations and hence higher processing overhead. It is important to note that the throughput performance also depends on the use of socket adaptors. The PF_RING-based LVRM generally has higher throughput than the raw-socket-based LVRM. As shown in Figure 4.2, if C++ VR is hosted, then the PF_RING-based LVRM outperforms the raw-socket-based LVRM for smaller frame sizes (e.g., by 50% when the frame size is 84 bytes). More importantly, it achieves very similar throughput as compared to the native Linux IP forwarding for all frame sizes.

We point out that there is room for further improving the achievable throughput of LVRM, for example, through the I/O optimization of the Linux network stack (e.g., see [13, 18]). Note that PF_RING is designed for packet capture, and it more optimizes the

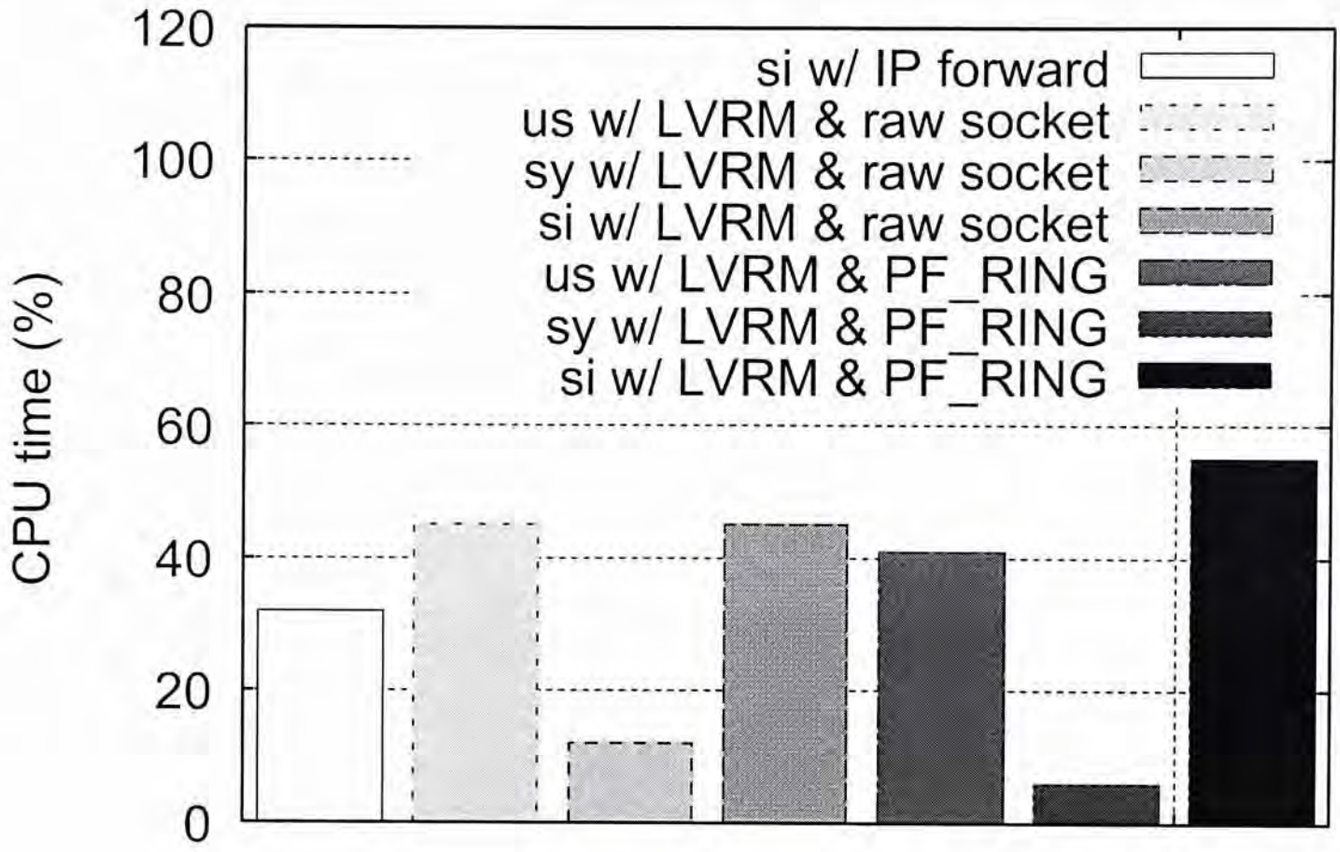

Figure 4.3: Experiment 1a: CPU usage in data forwarding.

receiving side of raw frames. On the other hand, optimizing the Linux network stack requires kernel modifications, and we pose this issue as future work.

Figure 4.3 shows the per-core CPU usages of different data forwarding mechanisms versus the frame size. In this experiment, we seek to show that LVRM in the user space (us) is not the key overhead with the minimum-sized frames in terms of the CPU time. We compare different data forwarding mechanisms as in the previous Experiment. To fully understand the internal CPU overhead of LVRM, we consider a different sets of CPU time e.g. the system CPU time (sy) and the software interrupts. Here, we consider only C++ VR and execute the system utility `top` in 'Batch mode' operation with 20 iterations.

First, we observe that the native Linux IP forwarding has the lowest overall CPU usage, servicing software interrupts only. This result is expected, since the data path is idle while waiting for incoming frames.

The overall CPU usages of the LVRM are higher than the native Linux IP forwarding. The reason is that in addition to handling the data frames within the kernel, they are also looping for the non-blocking polls from the sockets and the IPC queues, which are CPU-intensive. We observe that the raw-socket-based LVRM has higher CPU time in running kernel's and users' processes. Although we do not know the exact functions corresponding to software interrupts, but we intuitively believe shows that the CPU times may be spent in the x86 processors' assembly language instruction INT, for generating a software interrupt, which are typically executing kernel system calls.

It is important to note that only the minority of CPU times is in the user space. The PF_RING-based LVRM generally has lower user-space's CPU times than the raw-socket-based LVRM. As shown, LVRM achieves smaller CPU times within the code managed by us.

Experiment 1b (Round-trip latency in data forwarding). In this experiment, we seek to show that LVRM is not the key overhead compared to the network in terms of the latency of forwarding raw frames. We compare different data forwarding mechanisms as in Experiment 1a.

Figure 4.4 shows the results of different data forwarding mechanisms as defined in Experiment 1a. We observe that both Linux IP forwarding and different variants of LVRM return similar round-trip latencies, and their differences are mainly due to the variance in measurements. On the other hand, the general-purpose hypervisors QEMU-KVM and VMware Server return remarkably higher round-trip latencies.

Experiment 1c (Maximum achievable throughput with LVRM only). To fully understand the internal overhead (e.g., CPU or memory) of LVRM, we consider a different setting that excludes the network transmission part. Here, we load a trace file of 100M

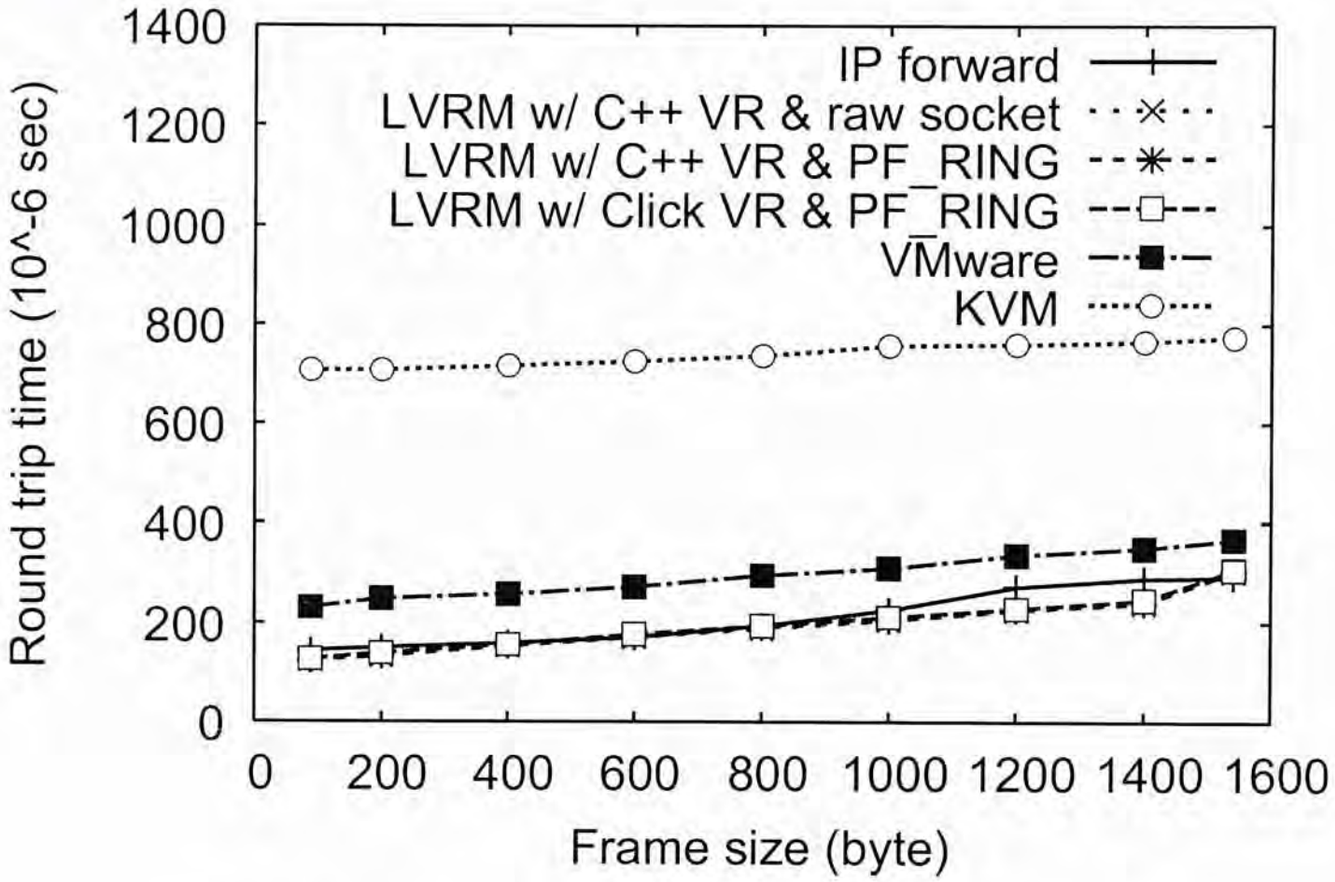

Figure 4.4: Experiment 1b: Round-trip latency in data forwarding.

minimum-sized frames (i.e., 84 bytes) into main memory within the gateway. We add an input interface to LVRM to read the raw frames from RAM, and add an output interface to LVRM to simply discard the frames. Then LVRM reads the frames from RAM as fast as possible, relays the frames to a hosted VR, and forwards the frames to the output interface that will simply discard the frames. This enables us to eliminate the overhead that occurs in network transmissions. Here, we consider both C++ VR and Click VR, both of which use a single VRI to process the frames.

Figure 4.5 shows the results. We note that C++ VR can achieve a significantly higher throughput than Click VR, mainly because the latter is the implementation of a software router and contains different internal operations that incur substantial processing overhead. Thus, the peak achievable throughput depends on the implementation of a VR. For C++ VR, which is a very simple VR implementation, LVRM can achieve 3.7M frames

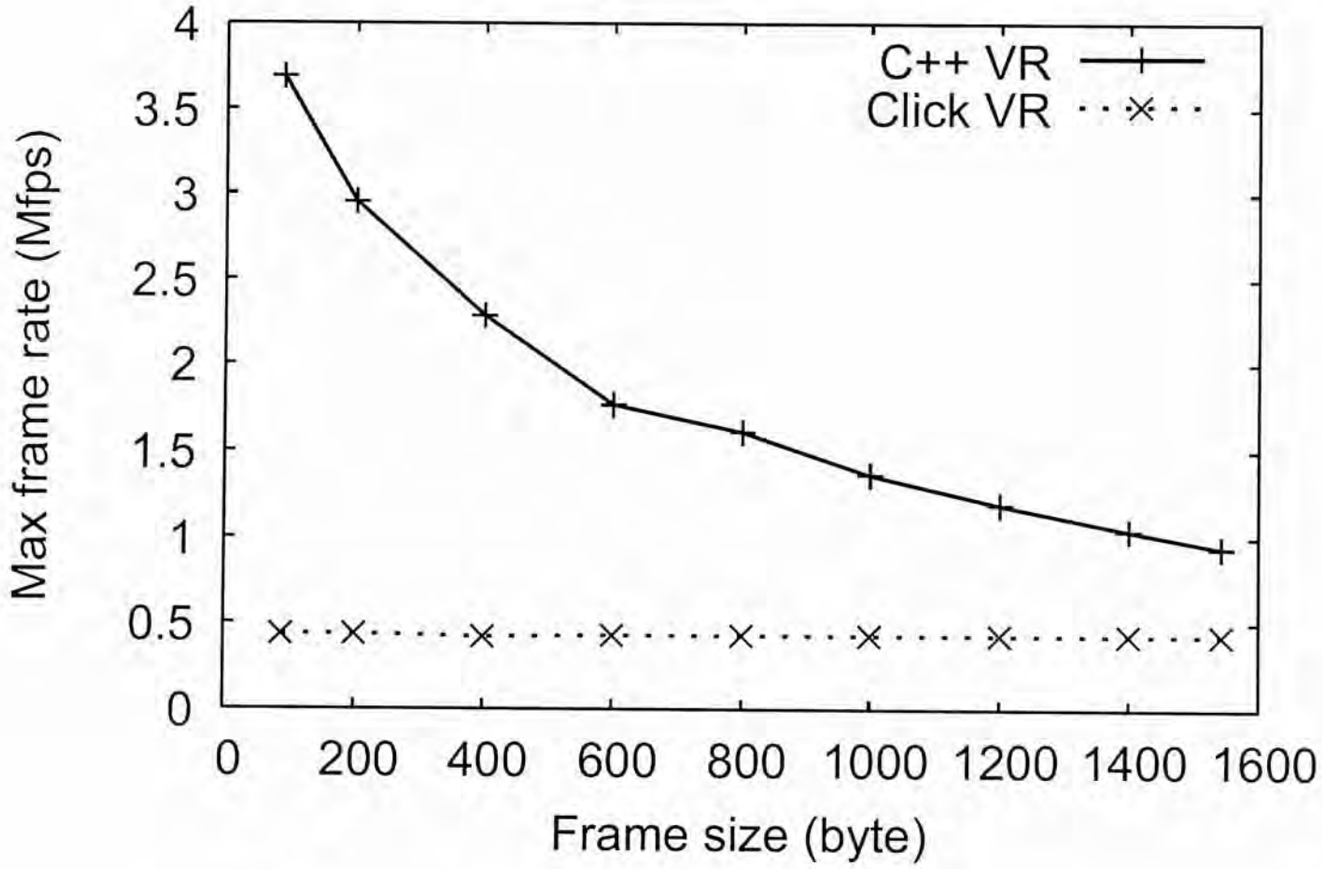

Figure 4.5: Experiment 1c: Achievable throughput with LVRM only.

per second for the smallest frame size 84 bytes; it can achieve 922K frames per second, or equivalently, 11Gbps, for the largest frame size 1538 bytes.

Experiment 1d (Round-trip latency with LVRM only). Similar to Experiment 1c, we evaluate the minimum round-trip latency with LVRM only by excluding the network transmission part. We use the same setting as in Experiment 1c, that is, we let LVRM read raw frames from main memory rather than from the network interface. LVRM forwards it to a VR that is hosted, and the VR forwards it to the output interface, where we simply discard the raw frames. We measure the latency of each frame from the input interface (i.e., main memory) to the output interface (i.e., where the raw frames are discarded) and compute the average latency for a given frame size.

Figure 4.6 shows the results. If C++ VR is hosted on LVRM, the latency is within 15 μs, as opposed to 70-120 μs as in Experiment 1b. Thus, LVRM by itself does not

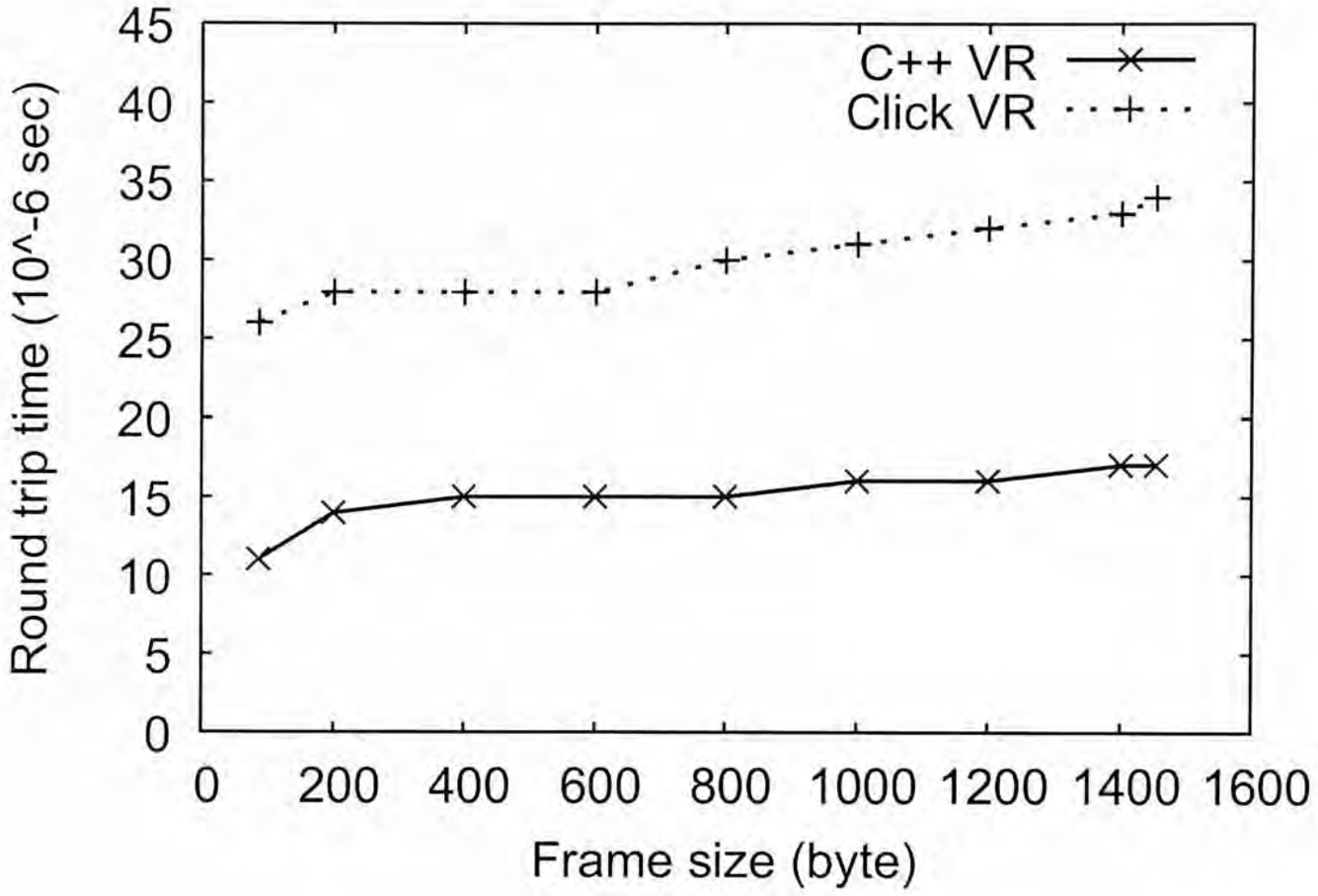

Figure 4.6: Experiment 1d: Round-trip latency with LVRM only.

contribute too much latency overhead as opposed to the network interface. The use of Click VR introduces a higher latency (in the range of 25-35 μs), but this latency remains small in general.

Experiment 1e (Latency of message passing). We now evaluate the latency of LVRM in relaying messages among VRIs. We have LVRM host a C++ VR, which has two VRIs. Then we have one of the VRIs send a control event to another VRI through the control queues. Then we measure the latency of such message passing between the two VRIs. We consider two settings: (i) *no load*, in which there is no raw data frames traversing LVRM, and (ii) *full load*, in which we use the topology in Figure 4.1 and have the sender hosts generate raw frames to the receiver hosts at the achievable throughput (see Experiment 1a).

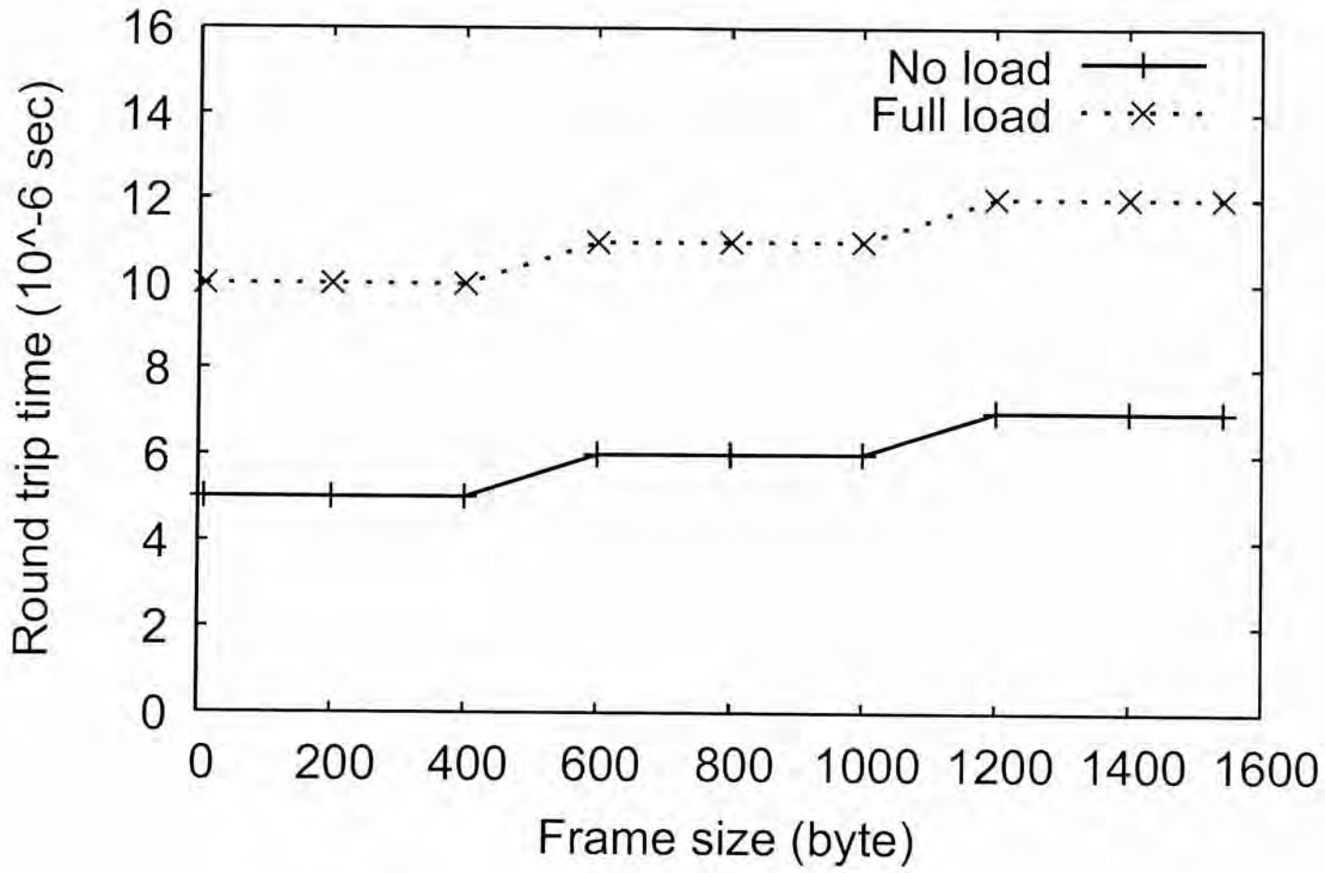

Figure 4.7: Experiment 1e: Latency of message passing.

Figure 4.7 shows the latency of relaying control events between two VRIs versus different sizes of the control events. The full-load setting has a higher latency than the no-load one. The reason is that in the full-load setting, a VRI is usually in the middle of processing a data frame when a control event arrives in the control queue, so it incurs some delay to retrieve the control event. However, we observe that the latency in the full-load setting remains in the range of 10-12 μs, which is relatively small compared to the network transmission part (see Experiment 1b). In the no-load setting, the latency is only in the range of 5-7 μs. Overall, the latency overhead of relaying control events between two VRIs is insignificant.

4.3 Core Allocation

We now evaluate the core allocation mechanism in LVRM. Based on the topology in Figure 4.1, we have the two sending hosts generate a certain traffic load, which contains raw frames of minimize frame size (i.e., 84 bytes), to the gateway on which we run LVRM. Our goal is to show that LVRM can dynamically allocate CPU cores to a VR based on the input traffic load.

Experiment 2a (Throughput analysis on core affinity). In this experiment, we evaluate how core affinity in the core allocation mechanism affects the throughput. We have LVRM host a single VR (either the C++ VR or the Click VR), and we create a single VRI for the VR to process raw frames. We run LVRM as a user-space process on a CPU core. Given that our gateway has two quad-core CPUs, we consider different approaches of allocating a CPU core for the VRI: (i) "sibling", in which LVRM dedicates a CPU core that resides in the same CPU as with LVRM, (ii) "non-sibling", in which LVRM dedicates a CPU core that resides in a different CPU as with LVRM, (iii) "default", in which LVRM lets the kernel assign the CPU core to the VRI, and (iv) "same", in which LVRM dedicates the same CPU core on which LVRM is currently running (i.e., it has two processes running on one core).

Figure 4.8 shows the achievable throughput for both C++ and Click VRs. Clearly, the "same" approach has the poorest performance, as a single core is bound with more than one process. For the Click VR, both the "sibling" and "non-sibling" approaches have similar achievable throughput, mainly because the bottleneck is due to the processing load of the Click implementation. However, for the C++ VR, we observe that the "sibling" approach has the highest achievable throughput. Thus, in general, it is more beneficial to first associate a sibling core to a VR if possible.

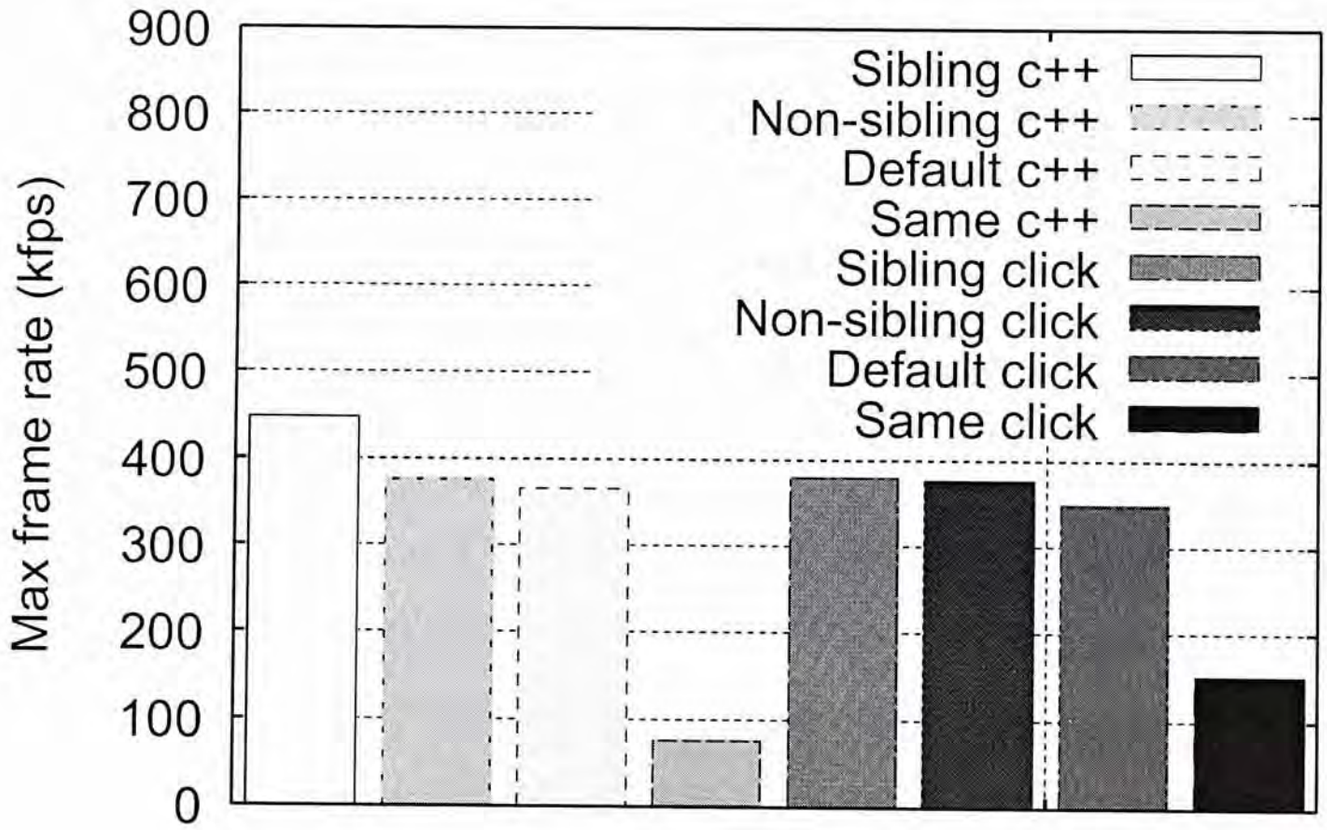

Figure 4.8: Experiment 2a: Throughput analysis on core affinity.

We also note that the "default" approach has less achievable throughput, even when compared to the "non-sibling" approach. The reason is that the kernel may occasionally switch a VRI process to a different core. This creates context switches, and will degrade the throughput performance. Thus, LVRM seeks to dedicate a core to a VRI process.

Experiment 2b (Throughput comparisons on fixed core allocation). In this experiment, we seek to show that it is important to adjust the number of cores assigned to a VR based on its traffic load. We have LVRM host a single VR (either the C++ VR or the Click VR). We then let LVRM fix the number of cores (i.e., VRIs) associated with the VR at the beginning when the VR is first started. Based on the topology in Figure 4.1, we inject a traffic load of maximum 360 Kfps. In the C++/Click VR implementation, we add a dummy processing load of 1/60 ms for each received raw frame before the raw frame is to be forwarded, in order to make the comparison of the various solutions in the

similar cases that the applications are CPU-bound. In the ideal case, if $c \leq 6$ cores are allocated for a VR, then the achievable throughput is $60c$ Kfps.

Figure 4.9 shows the achievable throughput of the C++/Click VR versus the number of cores allocated for the VR, as well as the maximum achievable throughput in the ideal case (labeled as "max"). Note that the gateway that we currently use has eight CPU cores, one of which is used by the LVRM process itself. Thus, we have seven cores available for the VR. We observe that the achievable throughput of the VR can scale up with the number of cores available. For the C++ VR, its achievable throughput is slightly less than the ideal case, implying that LVRM by itself is not a performance bottleneck. On the other hand, if the number of allocated cores is larger than the actual number of cores available in the gateway, then we observe contention, and the achievable throughput drops. Thus, LVRM seeks to limit the number of cores allocated for a VR based on the available CPU cores in the currently deployed system.

Experiment 2c (Dynamic core allocation). In this experiment, we evaluate the dynamic core allocation approach that adjusts the number of CPU cores based on the traffic load of a VR. We assume that LVRM hosts a single C++ VR, whose number of VRIs is varied by LVRM based on the current traffic load. Based on our topology in Figure 4.1, the two sending hosts generate an aggregate of traffic rate at S (in Kfps) for the C++ VR, while S increases from 60 to 360, and decreases from 360 to 60, at a step size of 60 at every 5 seconds. We also add a dummy processing load of 1/60 ms for each received raw frame to the VR implementation. We allocate c CPU cores to the VR if the aggregate traffic rate is $60(c-1)$ and $60c$ Kfps. For example, LVRM initially allocates one CPU core for the VR. If the aggregate traffic rate reaches the threshold 60 Kfps, then LVRM increments the number of cores for the VR to two. Note that each allocated core

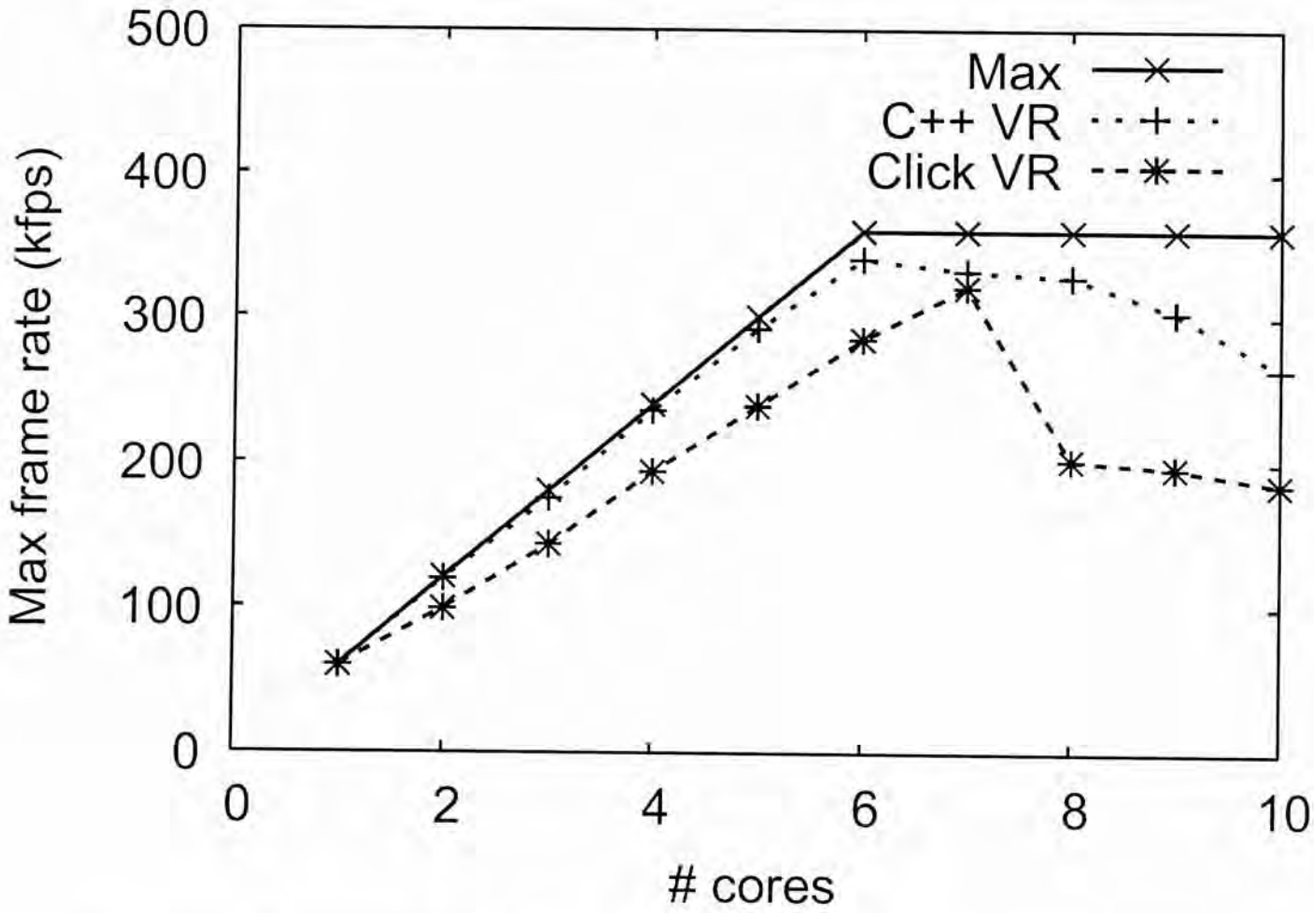

Figure 4.9: Experiment 2b: Throughput analyses on number of instances.

is associated with a VRI.

Figure 4.10 shows the number of CPU cores (or VRIs) allocated for the C++ VR with respect to the traffic rate. We observe that the number of cores is allocated for the VR in an expected manner. This shows that our dynamic core allocation approach can adapt the number of CPU cores with respect to the traffic load of a VR.

We also measure the time for allocating/deallocating a core for a VR (shown in the Figure 4.11), corresponding to the Figure 4.10. This latency metric corresponds to the time inclusively between the begin of iterating VR monitors and the end of creating/destroying a VR instance adapter, including the retrievals and the comparisons with the load estimates and the thresholds. In this Figure, we aim to show that the core reallocations of LVRM will *not* become a performance bottleneck in data forwarding latency. This Figure shows the reaction latency of different reallocations versus the elapsed time. In general,

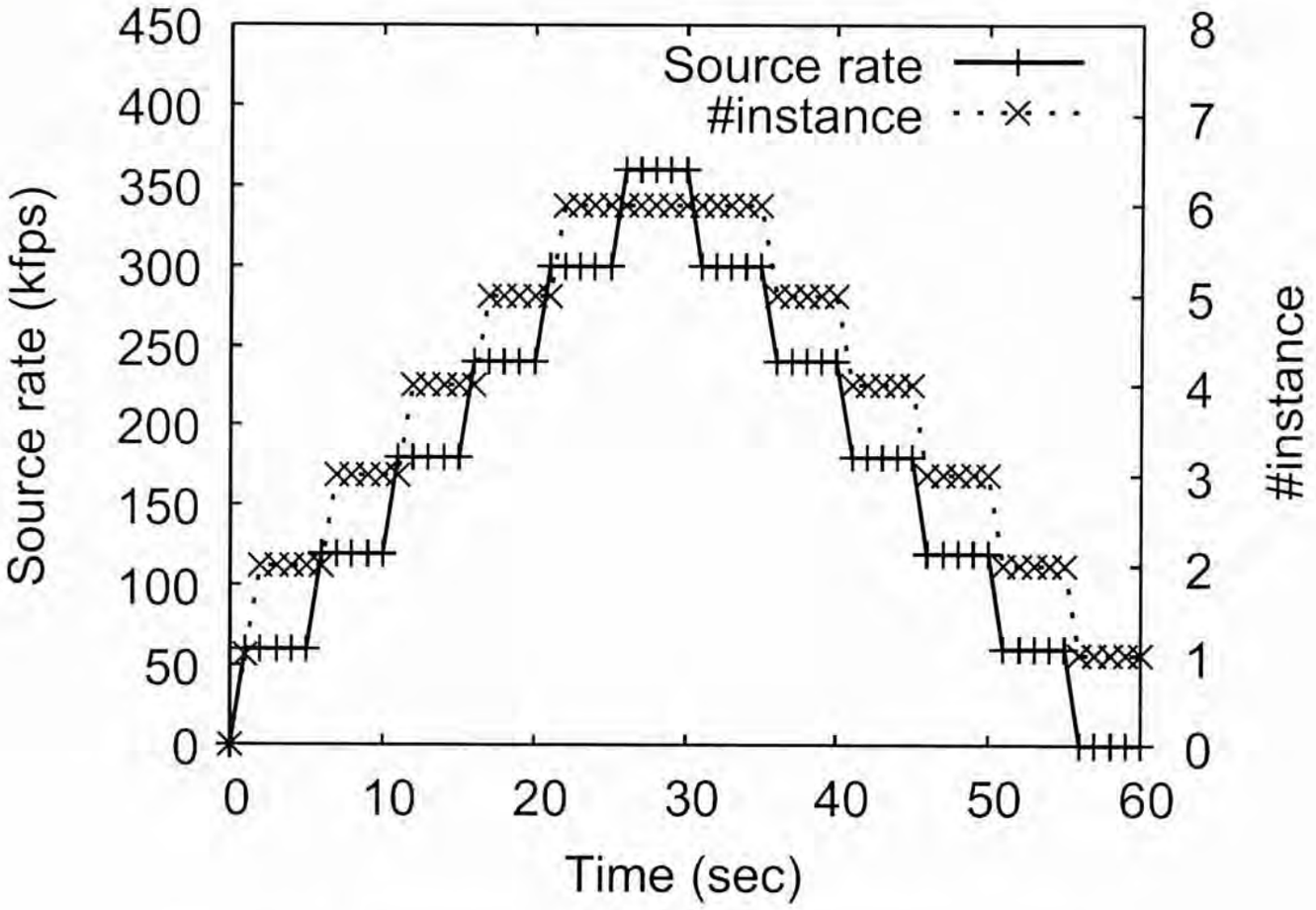

Figure 4.10: Experiment 2c: Dynamic core allocation for one VR.

we observe that the times for the allocations and the deallocations are within 900 μs and 700 μs, respectively. Thus, the core allocation/deallocation process can be completed within a small reaction time comparing to the International Telecommunication Union (ITU) standard G.114 [29] which states that 150 ms of one-way, end-to-end (from mouth to ear) delay ensures user satisfaction for telephony applications. As expected, the effect of LVRM to latency-sensitive applications such as Voice over Internet Protocol (VoIP) is not excessive. The reaction time is also small comparing to the TCP timeouts, and we show that the effect of LVRM to TCP is also not excessive in the Sections 4.4 and 4.5. Second, we observe that the allocations have the higher latencies than the deallocations. These results are expected, since the deallocations are simpler than the allocations involving the heavy-weight process creations.

The latency performance with more VRIs is slightly worse than that with less VRIs.

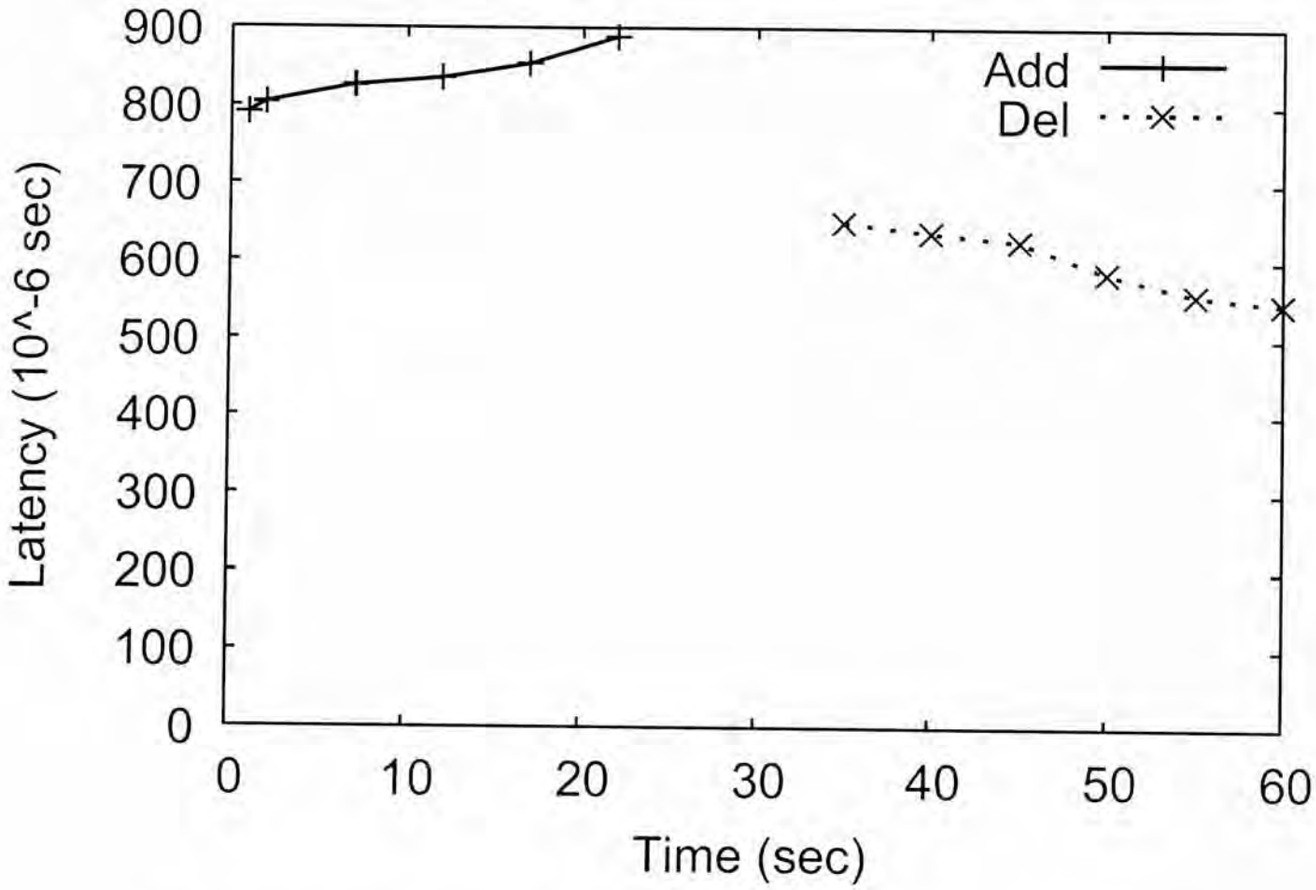

Figure 4.11: Experiment 2c: Dynamic core allocation for one VR.

The reason is that in addition to creating/destroying the VR instance adapter, iterating more VR monitors and retrieving more load estimates also incur additional performance overhead of accessing memories and processing various comparisons.

Experiment 2d (Dynamic core allocation with more than one VR). In this experiment, we aim to show that our dynamic core allocation can handle more than one VR. We now have LVRM host two C++ VRs. Based on our topology in Figure 4.1, each sending host generates a flow that is to be forwarded by a respective C++ VR. We also have the two flows start at different times. The core allocation condition is the same as in Experiment 2c, such that we allocate c CPU cores to each VR if the aggregate traffic rate is $60(c-1)$ and $60c$ Kfps. The traffic generation approach is similar to that in Experiment 2c, except that each flow has a maximum rate 180 Kfps and the step size is 30 Kfps.

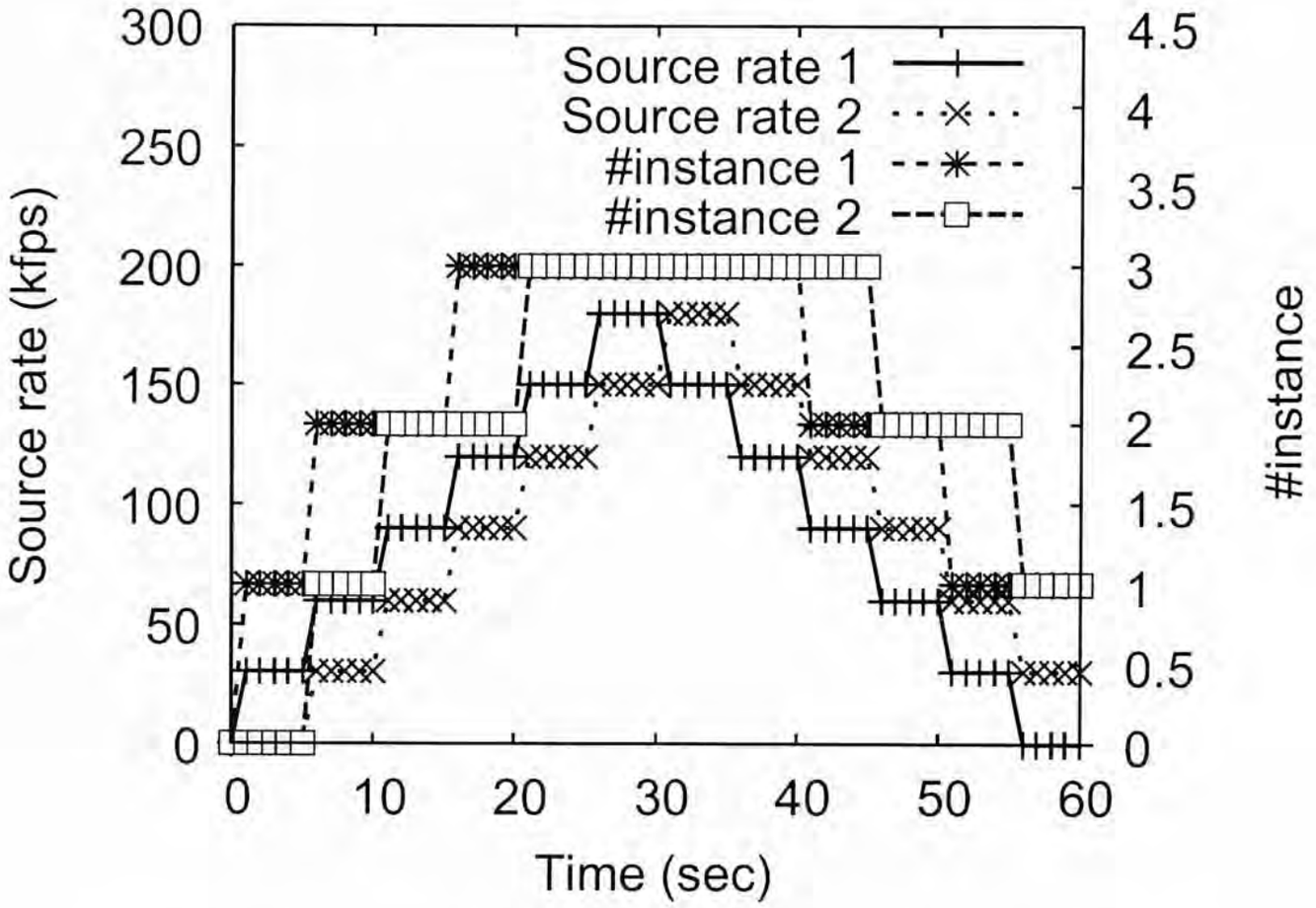

Figure 4.12: Experiment 2d: Dynamic core allocation for more than one VR.

Figure 4.12 shows how the core allocation scheme adjusts the numbers of CPU cores for each of the VRs based on their traffic rates. We observe that each of the VRs is allocated the number of cores in an expected manner, and the allocation is reflected with a small reaction time.

Experiment 2e (Dynamic core allocation with dynamic thresholds). In this experiment, we aim to show that our dynamic thresholds for the dynamic core allocation can handle VRs with different service rates. We also have LVRM host two C++ VRs. Based on our topology in Figure 4.1, each sending host generates a flow that is to be forwarded by a respective C++ VR. Nevertheless, we have the two flows start at the same time. The core allocation condition is similar to Experiment 2d, except that we allocate c CPU cores to each VR based on the dynamic thresholds. The traffic generation approach is similar to that in Experiment 2d, except that each flow has a maximum rate

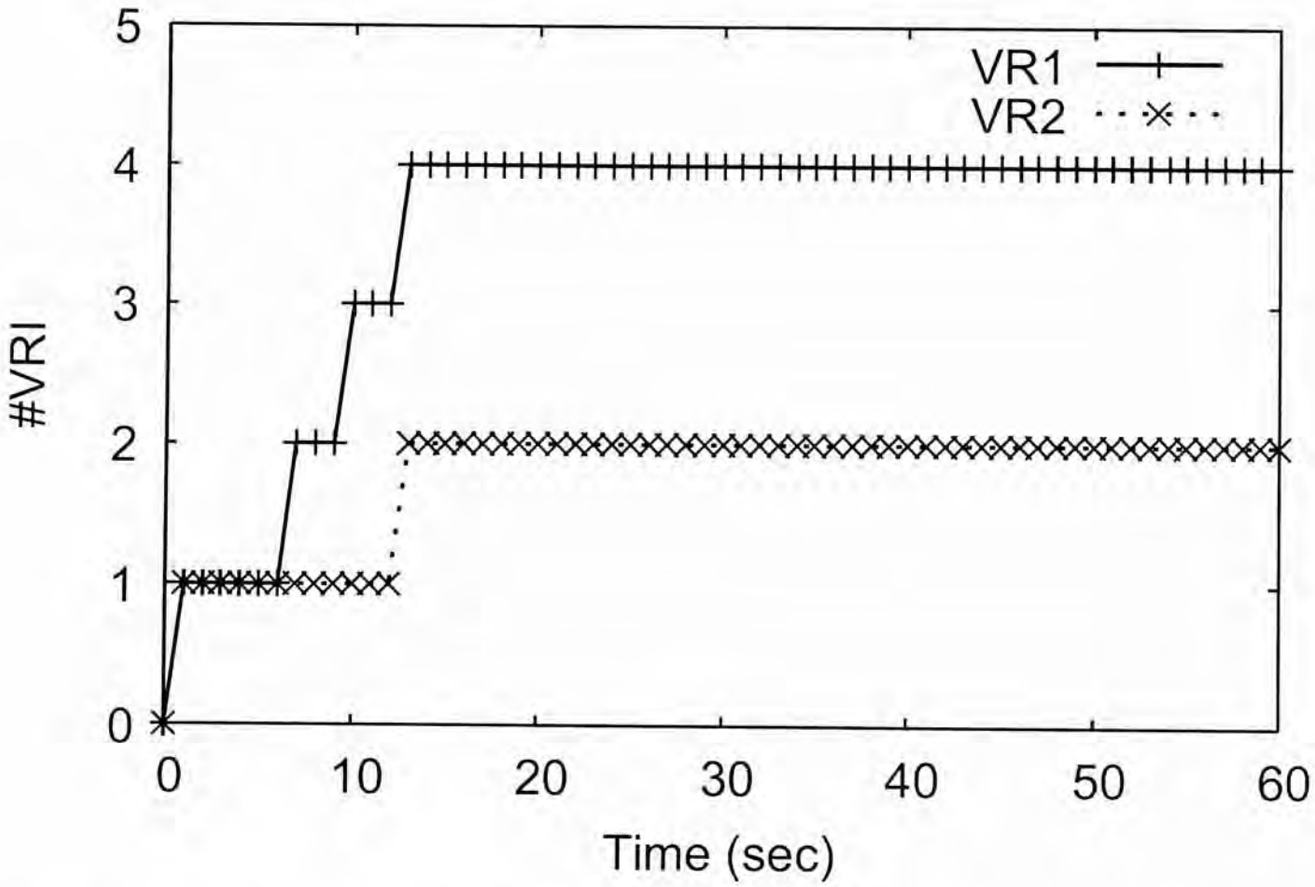

Figure 4.13: Experiment 2e: Dynamic core allocation with dynamic thresholds.

180 Kfps at the beginning and the ratio of VR1's to VR2's service rates is 1 : 2.

Figure 4.13 shows how the core allocation scheme adjusts the numbers of CPU cores for each of the VRs based on their service rates. We observe that each of the VRs is allocated the number of cores in an expected manner, and the allocation proportionally reflects the service times with a small error.

4.4 Load Balancing

In this section, we explore the load balancing implementation of LVRM. Similar to Section 4.3, we use the topology in Figure 4.1, we have the two sending hosts generate raw frames of minimize frame size (i.e., 84 bytes) to the gateway on which we run LVRM. Our goal is to explore the achievable throughput of different load balancing implementations.

Experiment 3a (Throughput of load balancing implementation in a single VR). In this experiment, we first evaluate how LVRM balances the processing load among VRIs of a single VR. We generate a traffic load of 360 Kfps to the gateway. We have LVRM host a single VR (either the C++ VR or the Click VR). In the VR implementation, we also add a dummy processing load of 1/60 ms to each VRI. Based on dynamic core allocation, the VR eventually is allocated six cores, each of which runs a VRI (see Experiment 2c). We evaluate different load balancing approaches, including join-the-shortest-queue (JSQ), round-robin (RR), and random (see Chapter 3). We then evaluate the achievable throughput of each load balancing scheme.

Figure 4.14 shows the achievable throughput of different load balancing schemes, as compared to the maximum achievable throughput (labeled "max") in the ideal case (i.e., 360 Kfps). The load balancing schemes have similar achievable throughput. The Click VR has less achievable throughput than the C++ VR, mainly due to its internal processing load. Note that JSQ slightly outperforms others since it distributes raw frames based on the current load of each VRI, while RR and random do not take this factor into account.

Experiment 3b (Load balancing among VRs). In this experiment, we evaluate how LVRM balances the loads of more than one VR. We have LVRM host two VRs that are either both C++ VRs or both Click VRs. Using the topology in Figure 4.1, each sending host generates a flow that is to be forwarded by a respective VR. The traffic rate of each flow is 180 Kfps (i.e., the aggregate traffic rate is 360 Kfps). For each of the two VRs, we measure the achievable throughput values (call them T_1 and T_2). Then we compute $T = 2 \times \min(T_1, T_2)$, and compare it with the ideal value (i.e., 360 Kfps). If T is close to the ideal value, then it implies that both VRs receive fair shares of processing load.

Figure 4.15 shows the results. For the C++ VR, we observe that the value of T for

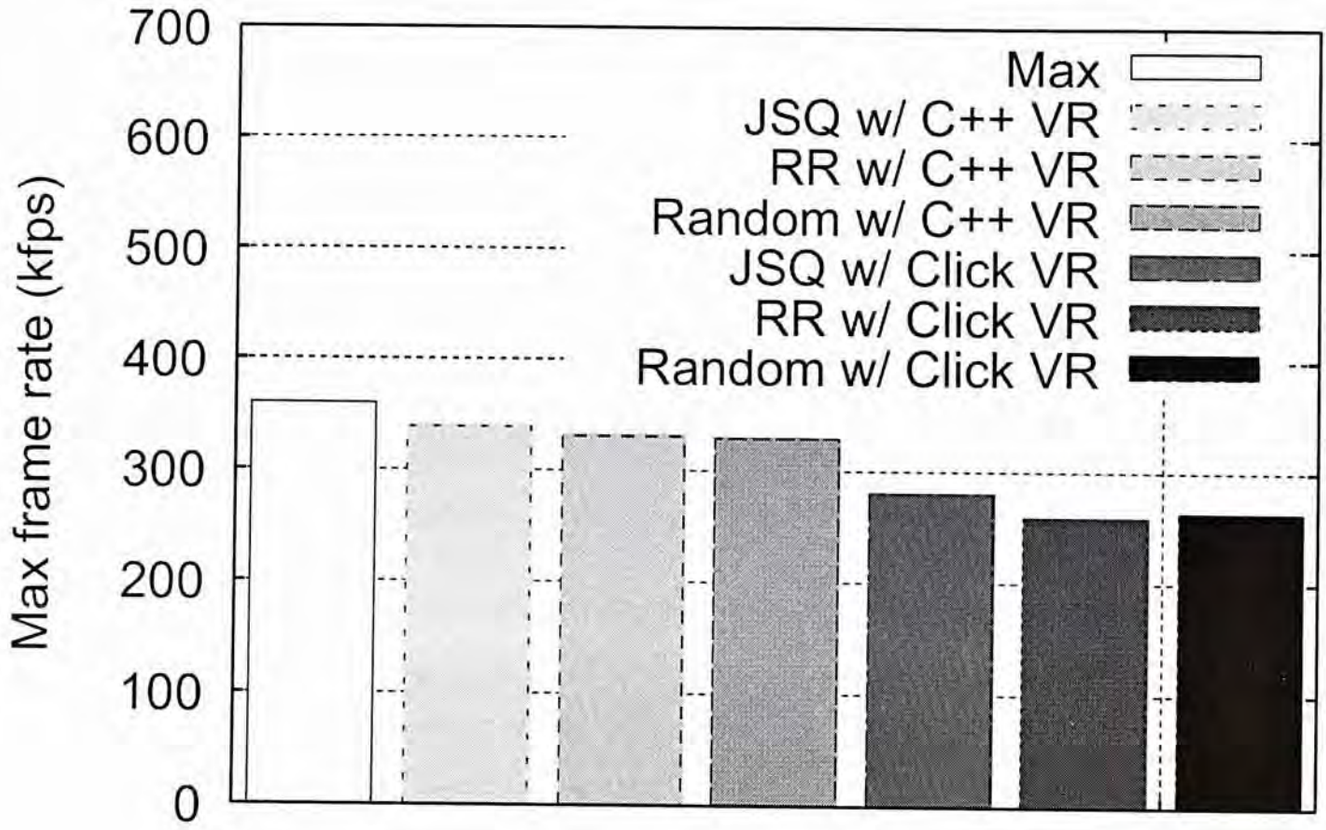

Figure 4.14: Experiment 3a: Load balancing among VRIs of a VR.

each of the load balancing scheme is very close to the ideal value (labeled as "Max"). For the Click VR, its achievable throughput is less due to its internal processing load. Overall, LVRM can maintain load balancing among more than one VR. Also, similar to Experiment 3a, we observe that JSQ outperforms other load balancing schemes.

Experiment 3c (Frame-based and flow-based load balancing). In this experiment, we evaluate how LVRM balances the FTP/TCP loads and compare both the frame-based and the flow-based load balancing. We have LVRM host at most six VRIs of the same VR that is C++ VR. Using the topology in Figure 4.1, it generates 100 pairs of flows that are to be forwarded by the VRIs (, respectively if the implementation is flow-based). Here, we consider two types of metrics: (i) the aggregate throughput and (ii) the fairness.

Figure 4.16 shows the aggregate throughputs of different load balancing. First, we

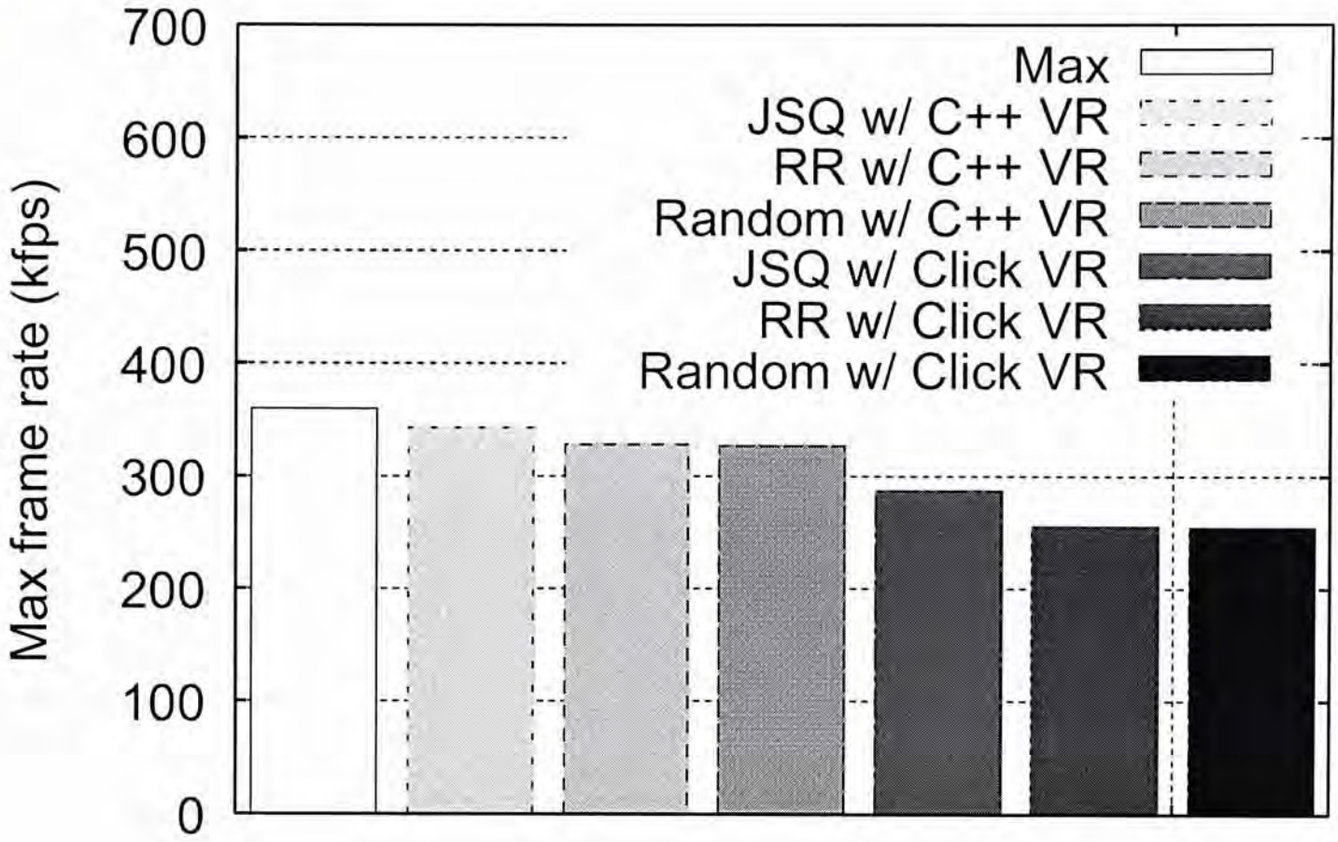

Figure 4.15: Experiment 3b: Load balancing among VRs.

observe that the native Linux IP forwarding and the LVRM with JSQ load balancing have the highest aggregate throughput, which are lower than the link rate. This result is expected, since TCP may push some small segments such as the FTP control messages and acknowledgements (see Section 4.2 for details of throughputs with small frames). It is reasonable that the aggregate throughputs are lower than the link rate.

The throughput performances of LVRM with flow-based load balancing are slightly worse than that with the frame-based load balancing, since the data paths with the flow-based load balancing are less simple such as involving some connection tracking. Besides accessing the hash tables, the connection tracking function also updates the timestamps of the flows by getting the process times via the system call `times()` which incurs some overheads.

Figure 4.17 shows the max-min fairness of different load balancing, normalized by their

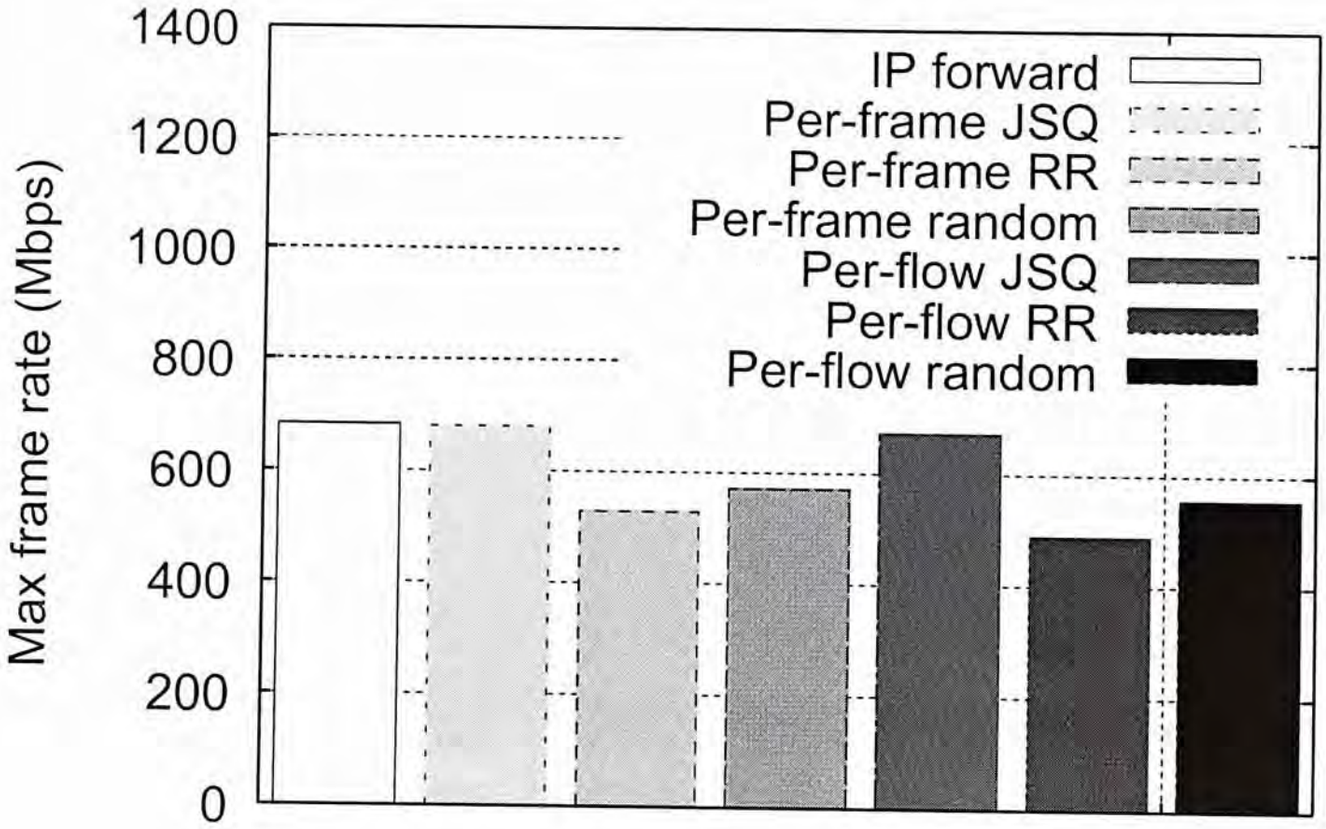

Figure 4.16: Experiment 3c: Aggregate throughputs.

corresponding aggregate throughputs. The indexes are all over 0.6, which are good. For LVRM with JSQ load balancing, it generally achieves the fairness as well as the native Linux IP forwarding does. We note that using flow-based load balancing has smaller fairness than using frame-based load balancing. The reason is that the flow-based load balancing is more sensitive to the variances of the flow sizes, as the granularity of the flow-based load balancing is coarser than that of the frame-based load balancing.

Figure 4.18 shows the Jain's fairness index of different load balancing. It is important to note that the indexes are all over 0.9, which are good. More importantly, it means that the majority of the flows are also fair with the use of LVRM with different load balancing.

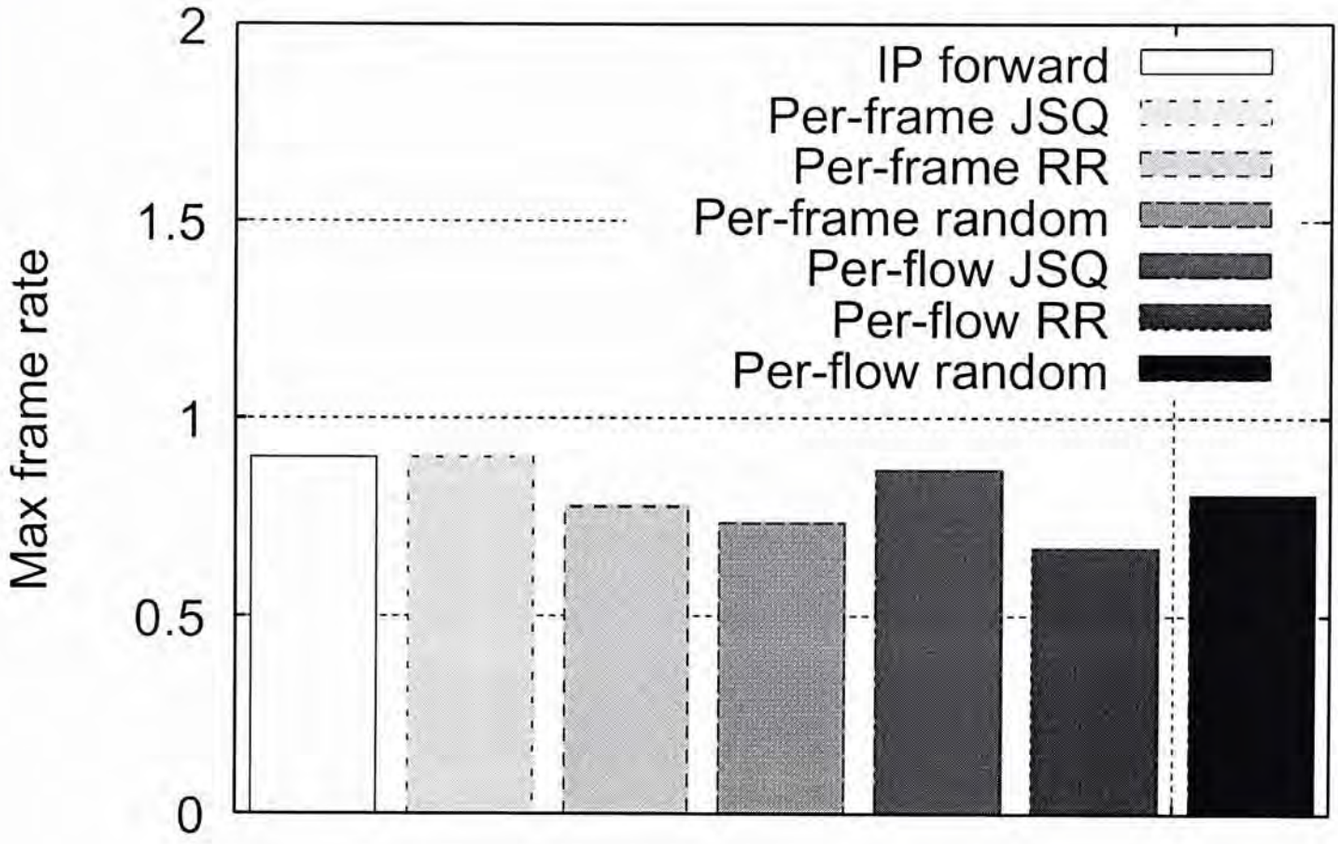

Figure 4.17: Experiment 3c: Max-min fairness.

4.5 Scalability

We finally evaluate the scalability of the flows in LVRM. We seek to address the following question:

- With only little losses of the throughputs and the fairness, is LVRM scalable in other complicated cases?

In this section, we consider the case where LVRM hosts at most six VRIs, and the VRIs used only process raw frames. Similar to Sections 4.3 and 4.4, it is based on the topology in Figure 4.1. We have four hosts generate realistic FTP traffic load, which contains TCP segments of various segment sizes, through the gateway on which we run LVRM. Our goal is to show that LVRM can maintain the throughputs and the fairness while it is scalable to large numbers of flows.

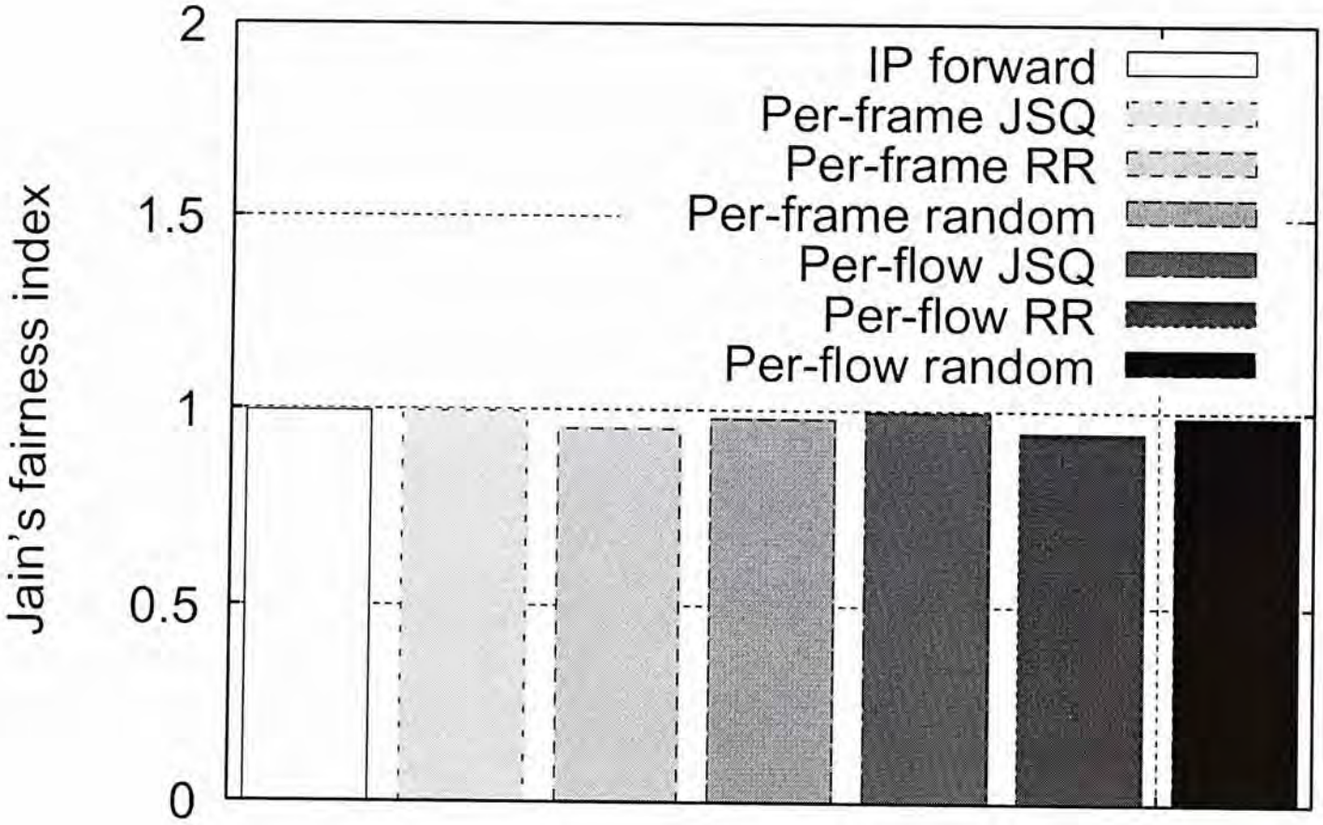

Figure 4.18: Experiment 3c: Jain's fairness index.

Experiment 4 (Scalability). In this experiment, we especially evaluate how the congestion control among TCP flows responds to LVRM. TCP generates a traffic load of almost maximum rate via the gateway. We have maximum of 1 Gbps (i.e., 1000 Mbps). In the VR implementation, we do not add a dummy processing load to each VRI as TCP responds to late segments. So mainly due to lost segments, the flows eventually have congestion crests, which are just below the maximum. We then evaluate the average rates in crests, aggregating different flows. We also evaluate the fairness of aggregate throughput. We use the max-min fairness and the Jain's fairness index of all flows.

Figure 4.19 shows the aggregate forward rate with different number of flows. The maximum aggregate rate in the ideal case is 1000 Mbps. The LVRM with frame-based load balancing and the native Linux IP forwarding have similar aggregate forward rate, which is slightly less than the ideal case. It is mainly due to TCP drops at maximum

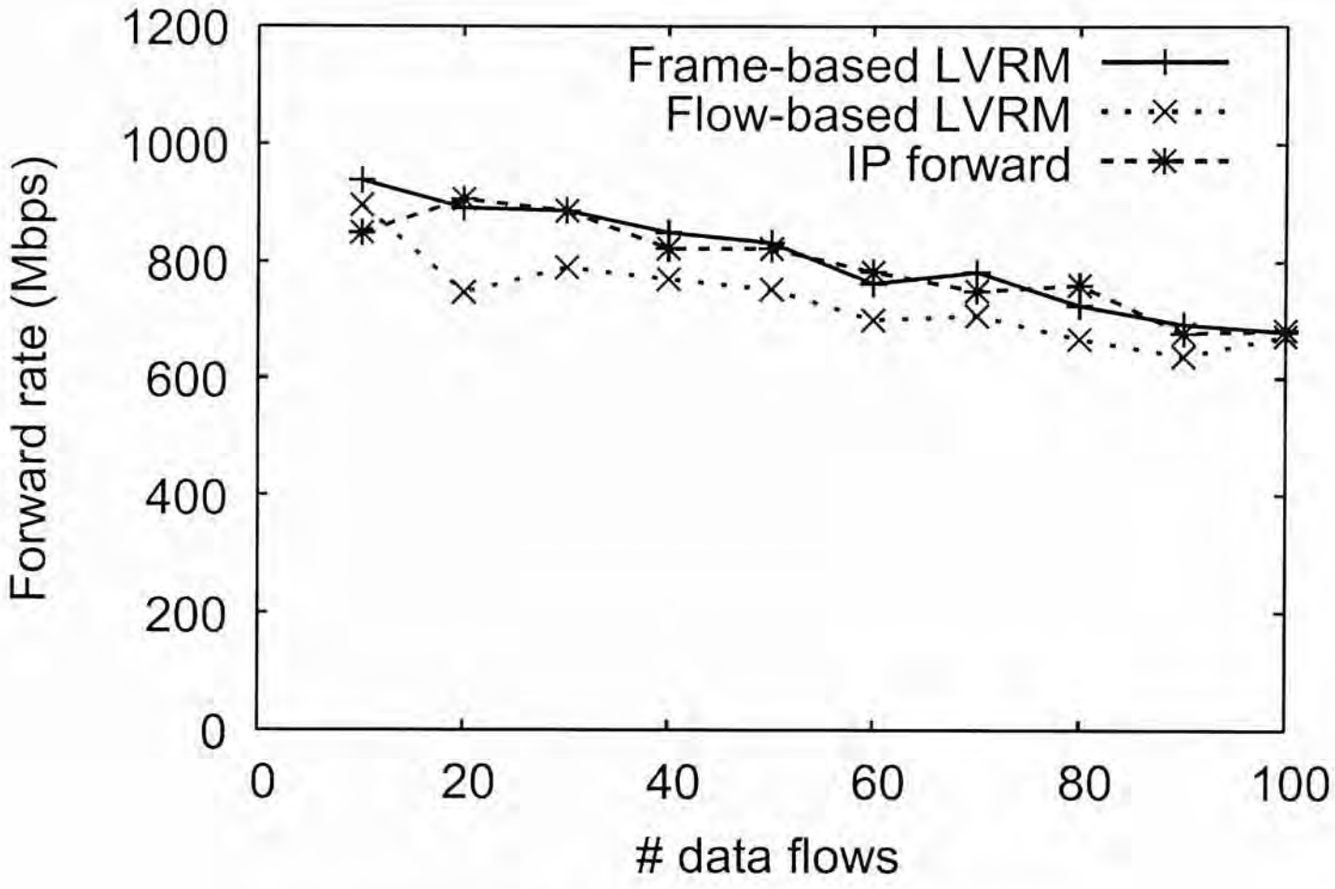

Figure 4.19: Experiment 4: Aggregate forward rate.

because of the congestion avoidance. Second, we also observe that the LVRM with frame-based load balancing has the higher aggregate forward rate for most numbers of flows. Similar to that in Experiment 3c, this result is expected. Overall, LVRM with more flows can still maintain the maximum rate.

Figure 4.20 shows the max-min fairness of different load balancing versus the number of data flow, normalized by their corresponding aggregate throughputs. The indexes are all over 0.8, which are very good. The fairness performances of the LVRM are roughly the same as that of the native Linux IP forwarding.

Figure 4.21 shows the Jain's fairness index of different load balancing versus the number of data flow. The indexes are all over 0.99, which are very good. For the Jain's fairness index, they generally achieve the almost highest fairness. We note that the majority of the flows are also fair with the use of LVRM with different load balancing.

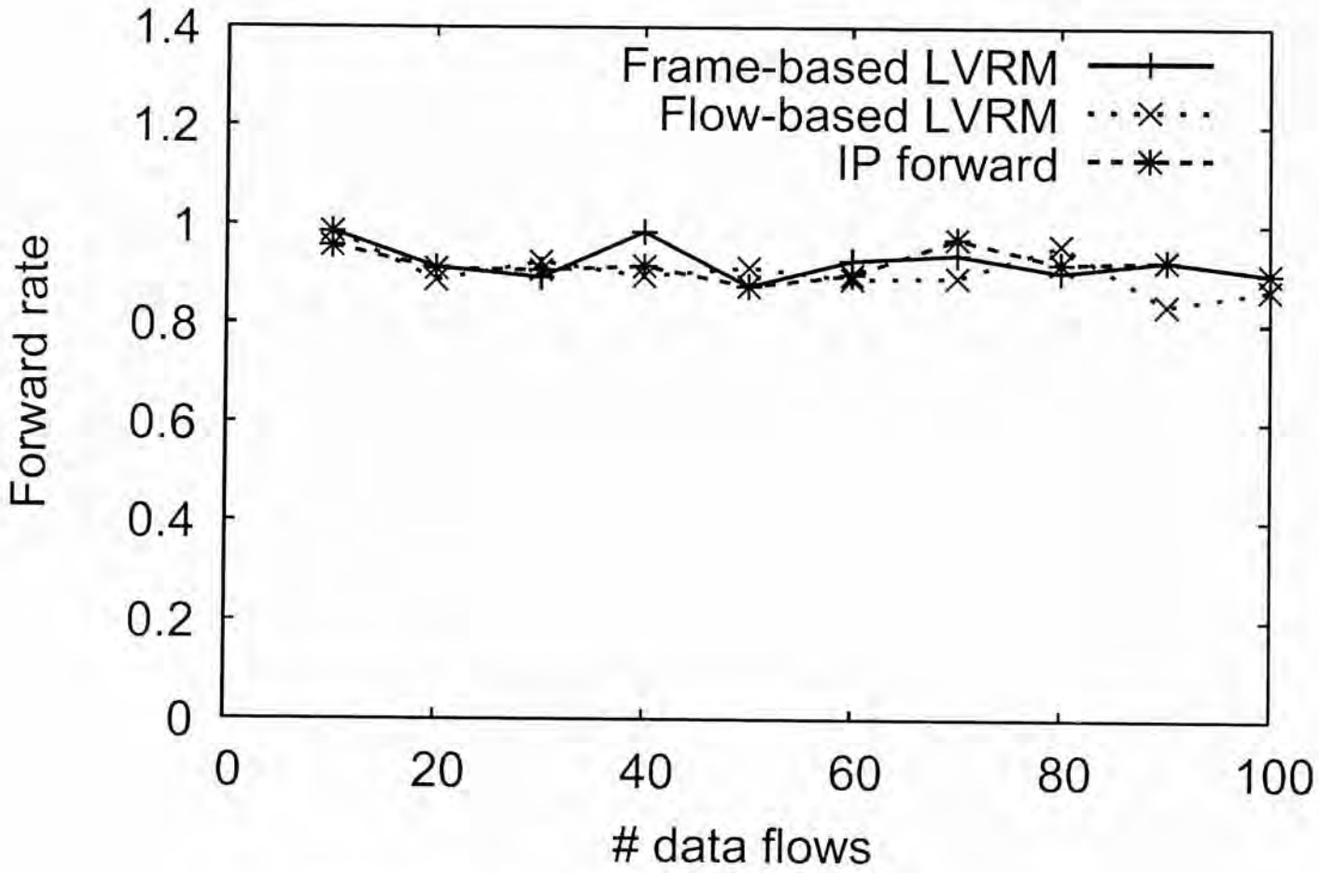

Figure 4.20: Experiment 4: Max-min fairness.

Figure 4.22 shows the aggregate forward of different load balancing with 100 pairs of flows versus the elapsed time. Most of the time, the forward rates are around 700 Mbps, which are good. It is also important to note that using LVRM has roughly the same forward rate as that of native Linux IP forwarding. For both the native Linux IP forwarding and the LVRM, there are little drops at the tails in terms of the throughput. We have to point out that the receive window of TCP's flow control also affect the source rate, since our realistic FTP programs read from the sockets and write to files but not simply discard the data. The results are that the source rates are sometimes faster than the rates of a FTP program to be scheduled by kernel in order to access the sockets and the files.

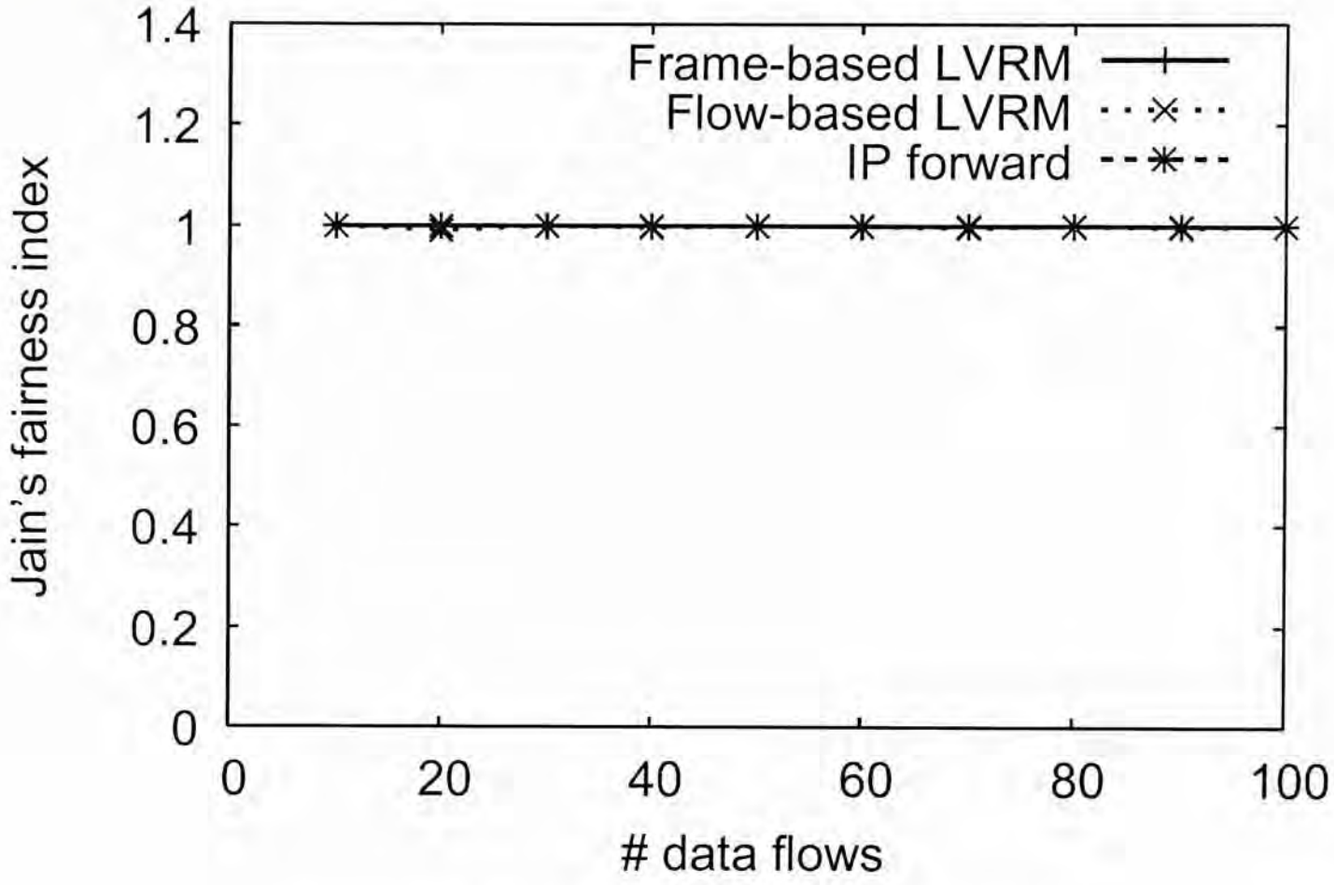

Figure 4.21: Experiment 4: Jain's fairness index.

4.6 Lessons Learned

We summarize the lessons learned from our experiments.

- Overall, LVRM itself incurs minimal performance overhead in data forwarding in terms of throughput and latency. It also provides a more lightweight approach than general-purpose hypervisors for hosting VRs.

- LVRM dynamically allocates CPU cores for VRs based on their traffic loads, with very small reaction times. To make core allocation effective, it is desirable to first select sibling cores that reside in the same CPU as LVRM for core allocation, and to dedicate a CPU core to at most one VRI.

- LVRM performs load balancing among VRIs of a VR, as well as among VRs. In general, the join-the-shortest-queue approach slightly outperforms other approaches

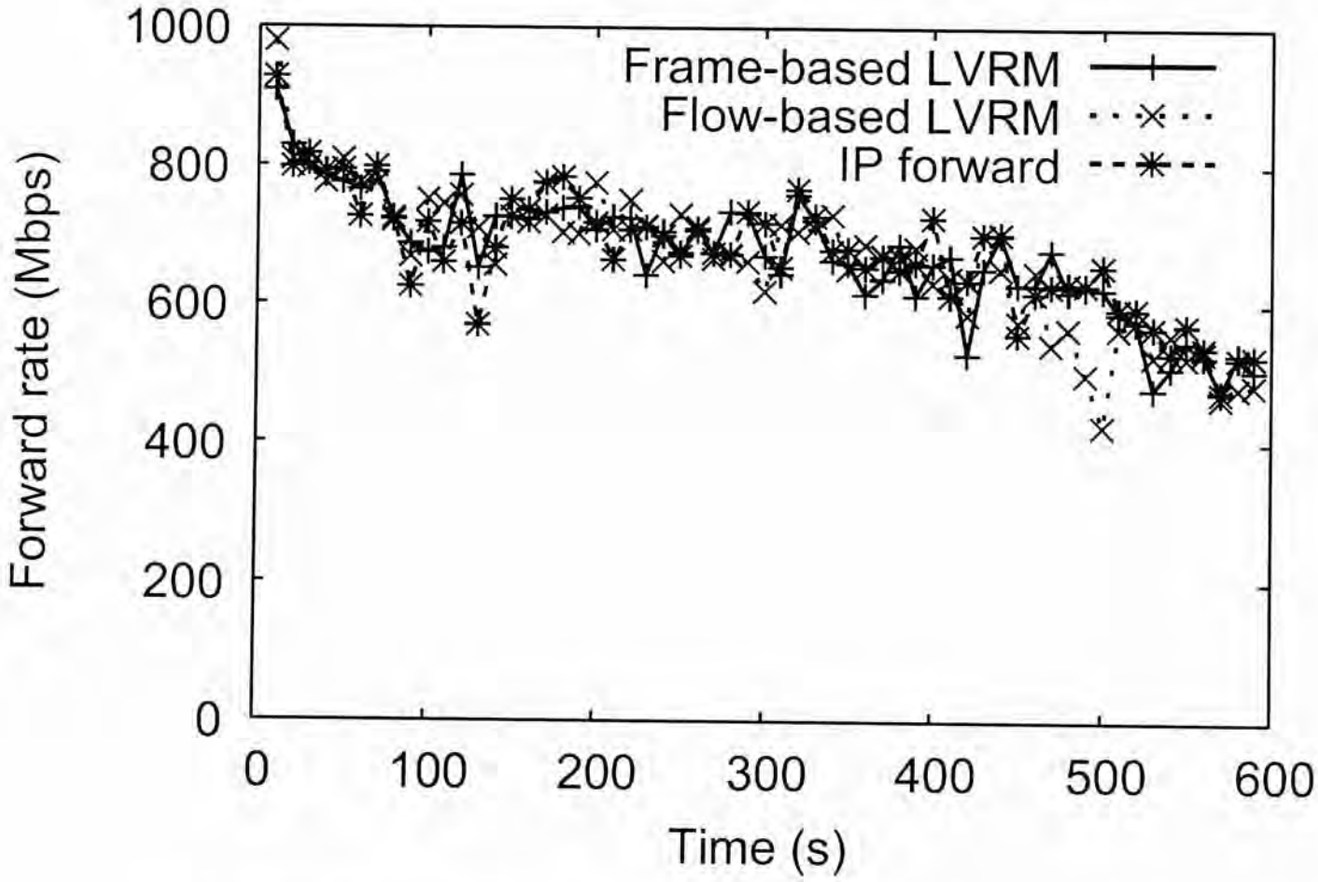

Figure 4.22: Experiment 4: Aggregate forward rate vs. elapsed time.

that do not consider the current load of a VRI, such as round robin and random.

- LVRM is scalable in other complicated cases. It also provides a fair approach as well as the native Linux IP forwarding.

Chapter 5

Related Work

Router virtualization has appeared in commercial products. For example, Cisco [9] and Juniper [22] partition the resources of a physical router into multiple logical routers, each of which has its own configuration and inventory information. However, their logical router management systems are not fully open-sourced, and hence lack the flexibility of customizing their resource management policies.

Software programmable routers have been studied for emulating routing functions of hardware routers. For example, router plugins [11] are proposed such that they can be dynamically configured, loaded, and unloaded in the kernel. Another software router architecture [27] is built on top of network processors, where low-level implementation issues of network processors are addressed. In particular, Click [21] and XORP [19] are extensible software router architectures that build configurable and flexible router instances and can run on commodity infrastructures. In the context of router virtualization, both Click and XORP can be extended for multi-process frameworks. SMP Click [6] proposes Click optimization on a multiprocessor setting that parallelizes packet processing, while XORP enables multiple routing processes for different routing protocols to run entirely

sandboxed. Other areas of router virtualization include router experimentation [3, 26], performance evaluation of software-based virtual routers on commodity hardware [14, 15], virtual router migration across hardware platforms [31], parallel executions of virtual machines on a single data plane [25], and network I/O fairness [1]. In particular, both [25, 1] address resource allocation of virtual instances in the context of network virtualization. In [25], it considers allocation of processing power among different forwarding engines; while in [1], it allocates an upper bound of bandwidth shares to virtual machines via rate limiting. Both of them do not consider how the allocation can be fine-tuned based on the load of each virtual instance.

With the emergence of multi-core technologies, multi-core router design is getting increasing attentions. A PC-based software router architecture [5] is proposed to use kernel-level enhancements (e.g., CPU core binding of kernel-level packet ring buffers) for a multi-core server to speed up the routing performance. RouteBricks [13] proposes a multi-core architecture that speeds up packet processing, and PacketShader [18] further accelerates packet processing using both multi-core and GPU technologies. Note that [13, 18] use available CPU cores for boosting the packet I/O performance, while we focus on using cores for packet processing inside routers. Our work differentiates itself from all the above virtual router architectures in that it considers CPU core allocation that is adaptive to the current traffic load.

Chapter 6

Conclusions

We explore the potential of building a router virtualization architecture in user space. We propose LVRM, a user-space load-aware virtual router monitor that hosts software-based virtual routers atop a commodity multi-core platform. A key feature of LVRM is to dynamically allocate CPU cores to different virtual routers based on their traffic loads. We propose an extensible design for LVRM that supports different variants of implementation including core allocation, load balancing, load estimation, and inter-process communication. We implement a proof-of-concept prototype of LVRM, and conduct extensive empirical experiments. We demonstrate that LVRM incurs minimal performance overhead in terms of throughput and latency as compared to hosting virtual routers atop general-purpose hypervisors. We also compare different variants of implementation for different components of LVRM, and show the extensibility of LVRM.

The source code of LVRM is published for academic use at `http://ansrlab.cse.cuhk.edu.hk/software/lvrm`.

Bibliography

[1] M. B. Anwer, A. Nayak, N. Feamster, and L. Liu. Network I/O Fairness in Virtual Machines. In *ACM SIGCOMM Workshop on VISA*, 2010.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. of ACM Symp. on Operating Systems Principles (SOSP)*, 2003.

[3] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *Proc. of ACM SIGCOMM*, 2006.

[4] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX ATC*, Apr. 2005.

[5] R. Bolla and R. Bruschi. PC-based Software Routers: High Performance and Application Service Support. In *ACM workshop on Programmable routers for extensible services of tomorrow*, Aug. 2008.

[6] B. Chen and R. Morris. Flexible control of parallelism in a multiprocessor PC router. In *USENIX ATC*, pages 333–346, June 2001.

[7] Cisco Systems, Inc. Bandwidth, packets per second, and other network performance metrics. http://www.cisco.com/web/about/security/intelligence/network_performance_metrics.html.

[8] Cisco Systems, Inc. Cisco 3700 Series Multiservice Access Routers [Cisco 3700 Series Multiservice Access Routers] - Cisco Systems. http://www.cisco.com/en/US/prod/collateral/routers/ps282/product_data_sheet09186a008009203f.html, 2004.

[9] Cisco Systems, Inc. Configuring Logical Routers on Cisco IOS XR Software. http://www.cisco.com/en/US/docs/ios_xr_sw/iosxr_r3.2/interfaces/configuration/guide/hc32logr.html, 2005.

[10] Cisco Systems, Inc. Cisco 7200 Series Routers - Products & Services - Cisco Systems. http://www.cisco.com/en/US/products/hw/routers/ps341/, 2011.

[11] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers. *ACM SIGCOMM Computer Communication Review*, 28(4):229–240, Oct 1998.

[12] L. Deri. Improving Passive Packet Capture: Beyond Device Polling. In *System Administration and Network Engineering*, Sept. 2004.

[13] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *ACM Symp. on Operating Systems Principles (SOSP)*, Oct 2009.

[14] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy. Towards High Performance Virtual Routers on Commodity Hardware. In *Proc. of ACM CoNEXT*, Dec. 2008.

[15] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, L. Mathy, and P. Papadimitriou. Implementing Software Virtual Routers on Multi-core PCs using Click. In *First Symp. on Click Modular Router*, Nov. 2009.

[16] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, L. Mathy, and T. Schooley. Evaluating Xen for Router Virtualization. In *Proc. of IEEE ICCCN*, 2007.

[17] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proc. of PPoPP*, 2008.

[18] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated Software Router. In *Proc. of ACM SIGCOMM*, 2010.

[19] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov. Designing Extensible IP Router Software. In *USENIX NSDI*, May 2005.

[20] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. *DEC Research Report TR-301*, Sept. 1984.

[21] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Trans. on Computer Systems*, 18(3):263–297, Aug. 2000.

[22] M. Kolon. Intelligent Logical Router Service. `http://www.juniper.net/solutions/literature/white_papers/200097.pdf`, Oct 2004. Juniper Networks, Inc.

[23] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Trans. on Software Engineering*, 3(2), Mar 1977.

[24] P. P. C. Lee, T. Bu, and G. Chandranmenon. A Lock-Free, Cache-Efficient Multi-Core Synchronization Mechanism for Line-Rate Network Traffic Monitoring. In *Proc. of IEEE IPDPS*, Oct 2010.

[25] Y. Liao, D. Yin, and L. Gao. PdP: Parallelizing Data Plane in Virtual Network Substrate. In *ACM SIGCOMM Workshop on VISA*, 2009.

[26] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, Apr 2008.

[27] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *ACM Symp. on Operating Systems Principles (SOSP)*, Oct 2001.

[28] W. R. Stevens. UNIX Network Programming: Networking APIs – Sockets and XTI, 1998.

[29] I. telecommunication Union. One-way transmission time. `http://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-G.114-200305-I!!PDF-E&type=items`, May 2003.

[30] VMware, Inc. VMware Server, Free Virtualization Download for Virtual Server Consolidation. `http://www.vmware.com/products/server/`, 2011.

[31] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, and J. Rexford. Virtual Routers on the Move: Live Router Migration as a Network-Management Primitive. In *Proc. of ACM SIGCOMM*, Jul 2008.