RouteBricks: Exploiting Parallelism To Scale Software Routers

Mihai Dobrescu[1] and Norbert Egi[2]*, Katerina Argyraki[1], Byung-Gon Chun[3], Kevin Fall[3], Gianluca Iannaccone[3], Allan Knies[3], Maziar Manesh[3], Sylvia Ratnasamy[3]

[1] EPFL
Lausanne, Switzerland

[2] Lancaster University
Lancaster, UK

[3] Intel Research Labs
Berkeley, CA

Abstract

We revisit the problem of scaling software routers, motivated by recent advances in server technology that enable high-speed parallel processing—a feature router workloads appear ideally suited to exploit. We propose a software router architecture that parallelizes router functionality both across multiple servers and across multiple cores within a single server. By carefully exploiting parallelism at every opportunity, we demonstrate a 35Gbps parallel router prototype; this router capacity can be linearly scaled through the use of additional servers. Our prototype router is fully programmable using the familiar Click/Linux environment and is built entirely from off-the-shelf, general-purpose server hardware.

1 Introduction

To date, the development of network equipment—switches, routers, various middleboxes—has focused primarily on achieving high performance for relatively limited forms of packet processing. However, as networks have taken on increasingly sophisticated functionality (*e.g.*, data loss protection, application acceleration, intrusion detection), and as major ISPs compete in offering new services (*e.g.*, video, mobility support services), there has been a renewed interest in network equipment that is programmable and extensible. In the absence of such extensibility, network providers have typically incorporated new functionality by deploying special-purpose network "appliances" or middleboxes [1, 8, 13–15]. However, as the cost of deploying, powering, and managing this assortment of boxes grows, the vision of a consolidated solution in the form of an extensible packet-processing "router" has grown more attractive. And indeed, both industry and research have recently taken steps to enable such extensibility [9, 17, 18, 40, 41].

The difficulty is that the necessary extensions often involve modification to the per-packet processing on a router's high-speed data plane. This is true, for example, of application acceleration [13], measurement and logging [8], encryption [1], filtering and intrusion detection [14], as well as a variety of more forward-looking research proposals [21, 36, 39]. In current networking equipment, however, high performance and programmability are often competing goals—if not mutually exclusive. On the one hand, high-end routers, because they rely on specialized and closed hardware and software, are notoriously difficult to extend, program, or otherwise experiment with. On the other, "software routers" perform packet-processing in software running on general-purpose platforms; these are easily programmable, but have so far been suitable only for low-packet-rate environments [16].

The challenge of building network infrastructure that is programmable *and* capable of high performance can be approached from one of two extreme starting points. One might start with existing high-end, specialized devices and retro-fit programmability into them [17, 18, 40, 41]. For example, some router vendors have announced plans to support limited APIs that will allow third-party developers to change/extend the software part of their products (which does not typically involve core packet processing) [17, 18]. A larger degree of programmability is possible with network-processor chips, which offer a "semi-specialized" option, *i.e.*, implement only the most expensive packet-processing operations in specialized hardware and run the rest on conventional processors. While certainly an improvement, in practice, network processors have proven hard to program: in the best case, the programmer needs to learn a new programming paradigm; in the worst, she must be aware of (and program to avoid) low-level issues like resource contention during parallel execution or expensive memory accesses [27, 32].

From the opposite end of the spectrum, one might start with software routers and optimize their packet-processing performance. The allure of this approach is that it would allow a broad community of developers to build and program networks using the operating systems and hardware platforms they tend to be most familiar with—that of the general-purpose computer. Such networks also promise greater extensibility: data and control plane functionality

*Work done while this author was an intern at Intel Labs Berkeley.

can be modified through a software-only upgrade, and router developers are spared the burden of hardware design and development. In addition, leveraging commodity servers would allow networks to inherit the many desirable properties of the PC-based ecosystem, such as the economic benefits of large-volume manufacturing, a widespread supply/support chain, rapid advances in semiconductor technology, state-of-the-art power management features, and so forth. In other words, if feasible, this could enable networks that are built and programmed in much the same way as end-systems are today. The challenge, of course, lies in scaling this approach to high-speed networks.

There exist interesting design points between these two ends of the spectrum. It is perhaps too early to know which approach to programmable routers is superior. In fact, it is likely that each one offers different tradeoffs between programmability and traditional router properties (performance, form factor, power consumption), and these tradeoffs will cause each to be adopted where appropriate. As yet however, there has been little research exposing what tradeoffs are achievable. As a first step, in this paper, we focus on one extreme end of the design spectrum and explore the feasibility of building high-speed routers using only PC server-based hardware and software.

There are multiple challenges in building a high-speed router out of PCs: one of them is performance; equally important are power and space consumption, as well as choosing the right programming model (what primitives should be exposed to the router's software developers, such that a certain level of performance is guaranteed as in a traditional hardware router). In this paper, we focus on performance; specifically, we study the feasibility of scaling software routers to the performance level of their specialized hardware counterparts. A legitimate question at this point is whether the performance requirements for network equipment are just too high and our exploration is a fool's errand. The bar is indeed high. In terms of individual link/port speeds, 10Gbps is already widespread; in terms of aggregate switching speeds, carrier-grade routers [5] range from 10Gbps up to 92Tbps! Software routers, in comparison, have had trouble scaling beyond the 1–5Gbps range [16].

Our strategy to closing this divide is RouteBricks, a router architecture that parallelizes router functionality across multiple servers *and* across multiple cores within a single server. Parallelization across servers allows us to incrementally scale our router capacity by adding more servers. Parallelizing tasks within a server allows us to reap the performance benefits offered by the trend towards greater parallelism in server hardware in the form of multiple sockets, cores, memory controllers, and so forth. We present RouteBricks' design and implementation, and evaluate its performance with respect to three packet-processing applications: packet forwarding, traditional IP routing, and IPsec encryption. We designed RouteBricks with an ambitious goal in mind—to match the performance of high-end routers with 10s or 100s of 1Gbps or 10Gbps ports. The results we present lead us to be cautiously optimistic about meeting this goal. We find that RouteBricks approaches our target performance levels for realistic traffic workloads, but falls short for worst-case workloads. We discover *why* this is the case and show that, fortunately, what is required to overcome this limitation is well aligned with current server technology trends.

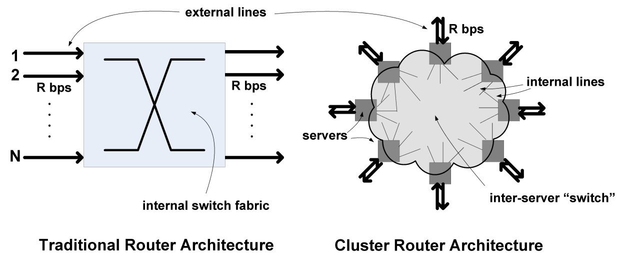

Figure 1: **High-level view of a traditional router and a server cluster-based router.**

We continue with a discussion of our guiding design principles and roadmap for the remainder of this paper.

2 Design Principles

Our ultimate goal is to make networks easier to program and evolve, and this leads us to explore a router architecture based on commodity, general-purpose hardware and operating systems. In this section, we summarize the design principles that emerged from translating this high-level goal into a practical system design.

Parallelism across servers. We want to design a router with N ports, each port with full-duplex line rate R bps. The role of the router is to receive the packets arriving at all these ports, process them, and transfer each incoming packet from its input port to the desired output port (which is typically determined by processing the packet's IP headers). This router's functionality can thus be broken into two main tasks: (1) packet processing, like route lookup or classification, and (2) packet switching from input to output ports. In current hardware routers, packet processing typically happens at the linecard, which handles from one to a few ports, while packet switching happens through a switch fabric and centralized scheduler; as a result, each linecard must process packets at a rate proportional to the line rate R, while the fabric/scheduler must switch packets at rate NR (*i.e.*, it must handle the aggregate traffic that traverses the router). Existing software routers, on the other hand, follow a "single server as router" approach; as a result, the server/router must perform switching *and* packet processing at rate NR.

In many environments, N and R can be fairly high. The most common values of R today are 1, 2.5 and 10Gbps, with 40Gbps being deployed by some ISPs; N can range from ten up to a few thousand ports. As specific examples: a popular mid-range "edge" router supports up to 360 1Gbps ports [3];

the highest-end "core" router supports up to 4608 10Gbps ports [5]. For such N and R, getting a general-purpose server to process and switch packets at rate NR is an unrealistic goal: such performance is 2–3 orders of magnitude away from current server performance and, even with recent server advances, we cannot hope to close so large a gap.

Recognizing this leads to our first design principle: that router functionality be parallelized across multiple servers, such that the requirements on each individual server can be met with existing or, at least, upcoming server models. This in turn leads us to a *cluster* router architecture (depicted in Fig. 1), where each server plays the role of a traditional router linecard, *i.e.*, performs packet processing for one up to a few router ports; as with a linecard, this requires that each server can process packets at a rate proportional to R. The problem then is how to switch packets between servers. We look for an appropriate *decentralized* solution within the literature on parallel interconnects [28]. In §3, we show that our use of commodity server and link technology narrows our options to solutions based on *load-balanced* interconnects, in which each node can independently make packet routing and dropping decisions for a subset of the router's overall traffic.

Thus, we design a router with no centralized components—in fact, no component need operate at a rate greater than cR, where c is independent of N and ranges typically from 2 to 3. By parallelizing both packet processing *and* switching across multiple servers, we thus offer an approach to building a router with N ports and line rate R bps, using servers whose performance need only scale with cR, independently of N.

Parallelizing router functionality across multiple servers also leads to an architecture that, unlike current network equipment, is incrementally extensible in terms of ports: For practical port counts (up to 2048 ports—shown in §3.3), we show that we can increase the number of ports by n (and switching capacity by nR) at a cost that scales linearly with n, simply by adding servers to the cluster. In this sense, a cluster of general-purpose servers is extensible, not only in terms of router functionality, but also in terms of capacity.

Parallelism within servers. Our cluster router architecture is only feasible if single-server performance can scale with cR. With 10Gbps ports and our lowest $c = 2$, this requires that a server scale to at least 20Gbps. While a less daunting target than NR, even this rate at first appears beyond the capabilities of commodity servers—recent studies [23, 30] report rates under 10Gbps, in the 1–4Gbps range for minimum-size packets. This leads us to our second design principle: that router functionality be parallelized not only across servers, but also across multiple processing paths within each server. In §5 we show that, although non-trivial, scaling to cR is within reach, provided: (1) the hardware architecture of the server offers internal parallelism that extends beyond processing power to I/O and memory accesses, and (2) the server's software architecture fully exploits this hardware parallelism. In §4.2, we discuss how such parallelism may be exploited.

Resulting tradeoff. The downside to our two key design decisions—using general-purpose servers and parallelizing router functionality—is that they make it difficult for our router to offer the strict performance guarantees that hardware routers have traditionally offered. In particular, packet reordering (considered undesirable due to its potential impact on TCP throughput) and latency are two fronts on which a software router may fundamentally have to offer more "relaxed" performance guarantees. We discuss performance in greater detail throughout this paper, but note here that the historical emphasis on strict performance guarantees, even for worst-case traffic workloads, arose more because of vendors' traditional benchmarking practices than the realism of these workloads or criticality of these performance guarantees. Considering this, and the fact that "relaxed" performance guarantees open the door to more general-purpose network infrastructure, we deemed this a tradeoff worth exploring.

In the remainder of this paper, we present the design and implementation of a parallel software router that embodies these principles: we describe how we parallelize router functionality across multiple servers in §3; how we design to exploit the parallelism within a server in §4; and finally evaluate our designs in §5-6.

3 Parallelizing Across Servers

In a cluster router, each server takes on packet processing for one or a few ports and also participates in cluster-wide distributed switching. The challenge in coping with packet processing is largely one of extracting the required performance from each server—our focus in §4. Here, we focus on designing a distributed switching solution that is compatible with our use of commodity servers.

3.1 The Problem

A switching solution serves two high-level purposes: (1) it provides a physical path connecting input and output ports and (2) it determines, at each point in time, which input gets to relay packets to which output port and, consequently, which packets are dropped. This must be achieved in a manner that offers the following guarantees: (1) 100% throughput (*i.e.*, all output ports can run at full line rate R bps, if the input traffic demands it), (2) fairness (*i.e.*, each input port gets its fair share of the capacity of any output port), and (3) avoids reordering packets. Hence, a switching solution involves selecting an interconnect *topology* with adequate capacity and a *routing* algorithm that selects the path each packet takes from its input to output port. In this, commodity servers limit our choices by introducing the following constraints:

1. *Limited internal link rates:* The "internal" links that interconnect nodes within the cluster (Fig. 1) cannot run at a rate higher than the external line rate R. This is because we want to use commodity hardware, including network interface cards (NICs) and link technology. *E.g.*, requiring an internal link of 100Gbps to support an external line rate of 10Gbps would be expensive, even if feasible.

2. *Limited per-node processing rate:* As motivated earlier, we assume that a single server can run at a rate no higher than cR for a small constant $c > 1$. We estimate feasible c values for today's servers in §5.

3. *Limited per-node fanout:* The number of physical connections from each server to other nodes should be a small value, independent of the number of servers. This is because we use commodity servers, which have a limited number of NIC slots. *E.g.*, today, a typical server can accommodate 4–6 NICs, where each NIC fits 2–8 ports.

We now look for an appropriate topology/routing combination within the literature on parallel interconnects [28]. An appropriate solution is one that offers the guarantees mentioned above, meets our constraints, and is low-cost. In the interconnect literature, cost typically refers to the capacity of the interconnect (*i.e.*, # links × link rate); in our case, the dominant cost is the number of servers in the cluster, hence, we look for a solution that minimizes this quantity.

3.2 Routing Algorithms

Design options. The literature broadly classifies interconnect routing algorithms as either single-path or load-balanced [28]. With the former, traffic between a given input and output port follows a single path. With *static* single-path routing, this path remains constant over time, independently of traffic demands. Such routing is simple and avoids reordering, but, to achieve 100% throughput, it requires that internal links run at high "speedups" relative to the external line rate R; speedups violate our first constraint, hence, we eliminate this option. The alternative is *adaptive* single-path routing, in which a centralized scheduler (re)computes routes based on current traffic demands—for instance, centralized scheduling of rearrangeably non-blocking Clos topologies.[1] Such centralized scheduling avoids high link speedups (as the scheduler has global knowledge and can pick routes that best utilize the interconnect capacity), but requires that the scheduler run at rate NR (it must read the state of *all* ports to arrive at a scheduling decision); this violates our second constraint, hence, we also eliminate this option.

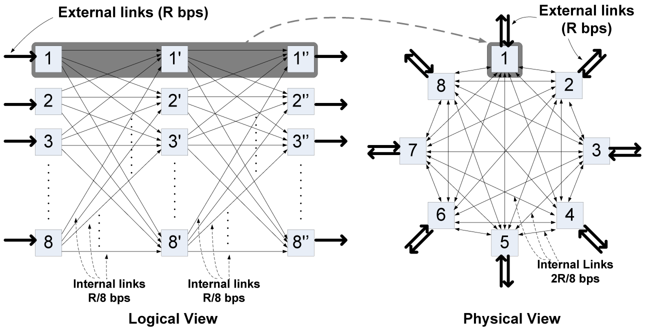

Figure 2: **An 8-port Valiant load-balanced mesh.**

We are thus left with *load-balanced routing*, where traffic between a given input and output port is spread across multiple paths. We start with a classic load-balancing routing algorithm—Valiant load balancing (VLB) [47]—and adapt it to our needs.

Background: VLB and Direct VLB. VLB assumes a set of nodes interconnected in a full mesh. Routing happens in two phases: consider a packet that enters at input node S and must exit at output node D; instead of going directly from S to D, the packet is first sent from S to a randomly chosen intermediate node (phase 1), then from that node to D (phase 2). In this way, a sequence of packets that enter at node S and exit at node D is first load-balanced from S across all nodes, then "re-assembled" at D. Fig. 2 shows an 8-node VLB mesh with each physical node playing the role of source, intermediary, and destination.

This routing algorithm has two desirable effects. Intuitively, phase-1 randomizes the original input traffic such that the traffic an individual node receives at the end of phase 1 (*i.e.*, the input traffic to phase 2) is a uniform sample of the *overall* input traffic at phase 1. As a result, when a node handles phase-2 traffic, it can make purely local scheduling decisions about which packets to drop and which to forward to each output port. This allows VLB to guarantee 100% throughput and fairness without any centralized scheduling. Second, VLB does not require high link speedups, because it forces traffic to be uniformly split across the cluster's internal links: in a full-mesh VLB cluster of N nodes with a per-port rate of R, each internal link must have capacity $\frac{2R}{N}$, which easily meets our constraint on internal link speeds.

These benefits come at the cost of forwarding packets twice. Without VLB, a server in a cluster-based router would be expected to handle $2R$ of traffic—R coming *in* from the server's external line (to be sent to the other servers) plus R arriving from the other servers to be sent *out* on the server's external line. With VLB, each node (as an intermediate) receives an additional R of incoming traffic and hence is required to process traffic at rate $3R$. Hence, the "tax" due to using VLB is a 50% increase in the required per-server processing rate.

[1] Recent work on data-center networks uses distributed scheduling over a rearrangeably non-blocking Clos topology, however, it does not guarantee 100% throughput [20].

"Direct VLB" [49] reduces VLB overhead by leveraging the following observation: Phase 1 in VLB serves to randomize traffic across the cluster; however, when the cluster's traffic matrix is *already* close to uniform (as is often the case), this first phase can be mostly avoided. More specifically, in "adaptive load-balancing with local information" [49], each input node S routes up to $\frac{R}{N}$ of the incoming traffic addressed to output node D directly to D and load-balances the rest across the remaining nodes. The authors show that this extension maintains the throughput and fairness guarantees of the original VLB. With this extension, when the cluster's traffic matrix is close to uniform, each server processes traffic at maximum rate $2R$, *i.e.*, VLB introduces no processing overhead.

So, VLB requires that each server process traffic at rate cR, where c is between 2 and 3, depending on the properties of the traffic matrix. We deem this as satisfying our second constraint (on server processing rates) and evaluate the extent to which today's servers can meet such processing rates in §5.

Our solution. Following the above reasoning, we start with Direct VLB, which allows us to guarantee 100% throughput and fairness, while meeting our constraints on link speeds and per-server processing rates. Two issues remain. First, like any load-balancing algorithm, VLB can introduce packet reordering. We address this with an algorithm that mostly avoids, but is not guaranteed to completely eliminate reordering; we describe it in §6, where we present our router prototype. The second issue is that our third constraint (on server fanout) prevents us from using a full-mesh topology when N exceeds a server's fanout. We address this by extending VLB to constant-degree topologies as described next.

3.3 Topologies

Design options. When each server handles a single router port, the lowest-cost topology is one with N servers as achieved by the full-mesh VLB. However, the full mesh becomes infeasible when N grows beyond the number of ports a server can accommodate (recall our constraint on per-server fanout).

One potential solution is to introduce intermediate nodes: rather than connect our N servers directly, connect them through extra nodes that form a low-degree multihop network. In this way, each server (both the N servers that handle the router ports and the extra, intermediate servers) only needs to run at rate $3R$ and can have low (even constant) fanout. Of course, this solution comes at the cost of an increase in cluster size.

Most multihop interconnect topologies fall under either the butterfly or the torus families [28]. We experimented with both and chose the k-ary n-fly (a generalized butterfly topology that interconnects N nodes with $n = \log_k N$ stages of k-degree nodes), because it yields smaller clusters for the practical range of parameters N and k that we considered (Fig. 3).

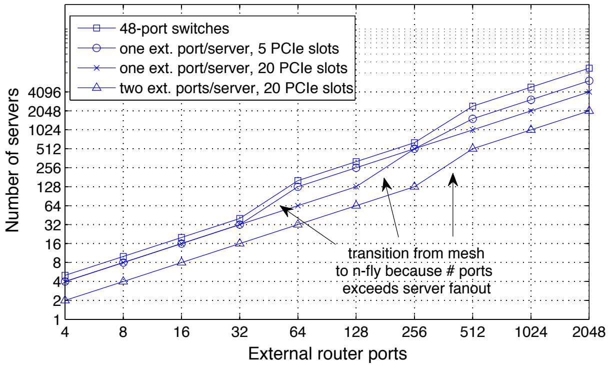

Figure 3: **The number of servers required to build an N-port, R=10Gbps/port router, for four different server configurations.**

If servers can run at speeds greater than $3R$, then we can also exploit the tradeoff between per-node fanout and processing rate: If the per-server processing rate is $3sR$ ($s > 1$), then each server can handle s router ports of rate R, hence, we need only $\frac{N}{s}$ input/output servers instead of N. In this case, building a full mesh requires a per-server fanout of $\frac{N}{s} - 1$ and internal link rates of $\frac{2sR}{N}$. Intuitively, if servers are more powerful, then we need fewer of them and fewer (but faster) links to interconnect them.

Our solution. We select a topology in the following way: First, we assign to each server as many router ports as it can handle (given its processing capacity and the port rate R). Next, we check whether the per-server fanout accommodates directly interconnecting the resulting number of servers in a full mesh. If not, we use a k-ary n-fly ($n = \log_k N$) topology, where k is the per-server fanout (as determined by the number of NIC slots and router ports per server).

We now look at the cost of our solution for three realistic scenarios: We consider a line rate of R = 10Gbps. We assume that each NIC accommodates 2 10Gbps or 8 1Gbps ports (the latter is currently available in compact form-factor cards). We consider three configurations:

1. *Current servers:* Each server can handle one router port and accommodate 5 NICs.

2. *More NICs:* Each server can handle one router port and accommodate 20 NICs. Such servers are available today as a custom motherboard configuration (*i.e.*, requiring no component redesign), typically for data-centers.

3. *Faster servers with more NICs:* Each server can handle two router ports and accommodate 20 NICs. This configuration corresponds to expected upcoming servers (§5).

For each scenario, Fig. 3 plots the total number of cluster servers, N', required to handle N external ports, as a function of N. Ignoring, for the moment, the results labeled "48-port switches," we see that, with the "current server" configuration, a full mesh is feasible for a maximum of $N = 32$ external ports; the "more NICs" configuration extends this to $N = 128$ ports, and the "faster servers" configuration to $N = 2048$ ports. We conclude that our cluster architecture can scale to hundreds, potentially even a few thousand ports. We note that, in theory, our design can always scale to an arbitrary number of external ports by using extra intermediate servers. However, in practice, there will always be an upper limit determined by the increase in cluster size that leads to higher per-port cost and power consumption, as well as higher per-packet latency. The results in Fig. 3 represent a range of port counts for which we believe these overheads to be reasonable. For example, even with current servers, we need 2 intermediate servers per port to provide $N = 1024$ external ports (a 10Tbps router). Assuming each server introduces 24μsec of latency (§6.2), such a configuration would correspond to 96μsec of per-packet latency.

Switched cluster: a rejected design option. Fig. 3 also shows the cost of an alternative, "switched" cluster that we considered early on. In this architecture, packet processing is performed by general-purpose servers, whereas switching is delegated to commodity Ethernet switches: for a low N, servers are interconnected by a single switch; when N exceeds the port count of a single switch, servers are interconnected through a network of switches, arranged in a strictly non-blocking constant-degree Clos topology [11].

We ultimately rejected this option for two reasons. First, guaranteeing switching performance using a network of switches requires support for new load-sensitive routing features in switches; such modification is beyond our reach and, even if adopted by switch vendors, would be significantly more complex than the simple load-agnostic routing switches currently support. Second, a back-of-the-envelope calculation reveals that, considering current products, the switched cluster would be more expensive: We considered a 48-port 10Gbps/port Arista 7148S switch, which, at $500 per port, is the least expensive strictly non-blocking 10Gbps switch we are aware of. Assuming $2000 per server, 4 Arista ports correspond to 1 server. Using this "conversion rate," we computed the number of servers whose aggregate cost is equal to an Arista-based switched cluster, and plotted this number as a function of the number of external ports N.

Fig. 3 shows the result, which indicates that the Arista-based switched cluster is more expensive than the server-based clusters. For small numbers of ports where we can interconnect servers in a full mesh, the server-based cluster is lower cost, because it avoids the cost of the switch altogether while using the same number of servers. For higher port counts, the difference in cost is due to the significant level of over-provisioning that a non-blocking interconnect must provide to accommodate non-uniform traffic matrices; our RouteBricks architecture avoids this through the use of VLB. The penalty, of course, is that the latter requires a per-server processing rate of $3R$, whereas a switched-cluster architecture requires a per-server processing rate of $2R$.

Summary. We select a router architecture that parallelizes both packet processing and switching over a VLB interconnect built from general-purpose servers. Our architecture relies on two key assumptions. First, that a modern server can handle at least one router port of rate R. Specifically, since we are using Direct VLB, to handle one port, a server must process packets at a *minimum* rate of $2R$ (assuming a uniform traffic matrix) or $3R$ (assuming a worst-case traffic matrix). For a line rate of $R = 10$Gbps, these requirements become 20Gbps and 30Gbps, respectively. The second (and related) assumption is that a practical server-based VLB implementation can live up to its theoretical promise. Specifically, VLB's requirement for a per-server processing rate of $2R$–$3R$ is derived assuming that all VLB phases impose the same burden on the servers; in reality, certain forms of processing will be more expensive—*e.g.*, IP route lookups *vs.* load-balancing. Thus, we need to understand whether/how VLB's analytically derived requirement deviates from what a practical implementation achieves.

The following sections test these assumptions: in §4 and §5 we evaluate the packet-processing capability of a state-of-the-art server; in §6, we present and evaluate RB4, a 4-node prototype of our architecture.

4 Parallelizing Within Servers

According to the last section, assuming a line rate of $R = 10$Gbps, our architecture is feasible only if each server can meet a minimum performance target of 20Gbps. This is more than twice the rates reported by past studies. Yet recent advances in server hardware technology promise significant speedup for applications that are amenable to parallelization; to leverage these advances, router software must exploit hardware resources to their fullest. Thus, in this section, we look for the right approach to parallelizing packet processing *within* a server. We start with an overview of our server architecture (§4.1), then discuss how we exploit server parallelism (§4.2).

4.1 Server Architecture

Hardware. For our study, we chose an early prototype of the Intel Nehalem server [19], because it implements the most recent advances in server architecture, while conforming to the informal notion of a "commodity" server (it is targeted to replace the currently deployed generation of Xeon servers). Fig. 4 shows this architecture at a high level: There are multiple processing cores,[2] arranged in "sockets"; all

[2] We use the terms "CPU," "core," and "processor" interchangeably.

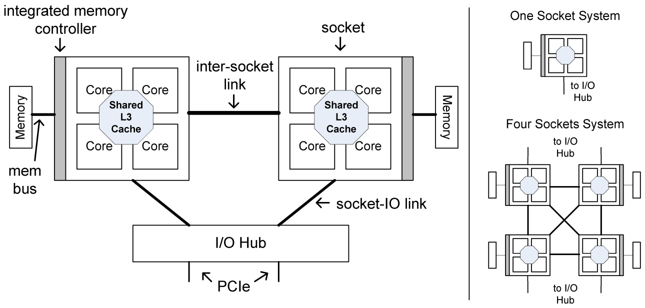

Figure 4: **A server architecture based on point-to-point inter-socket links and integrated memory controllers.**

cores in a socket share the same L3 cache. Each socket has an integrated memory controller, connected to a portion of the overall memory space via a memory bus (hence, this is a non-uniform memory access—NUMA—architecture). The sockets are connected to each other and to the I/O hub via dedicated high-speed point-to-point links. Finally, the I/O hub is connected to the NICs via a set of PCIe buses. Our server has two sockets, each with four 2.8GHz cores and an 8MB L3 cache, and two PCIe1.1 x8 slots, which we populated with two NICs, each holding two 10Gbps ports [6]. This server is the dual-socket configuration of the single-socket Corei7 server, and future configurations are expected to incorporate both more sockets (Fig. 4) and more cores per socket, as well as 4–8 PCIe2.0 slots.

Software. Our server runs Linux 2.6.19 with Click [38] in polling mode—*i.e.*, the CPUs poll for incoming packets rather than being interrupted. We started with the Linux 10G Ethernet driver, which we extended as described in §4.2. We instrumented our server with a proprietary performance tool similar to Intel VTune [7].

Traffic generation. We equipped our server with two dual-port 10Gbps NICs, hence, a total of four 10Gbps ports. However, we are only able to drive each NIC at 12.3Gbps: each pair of ports on the same NIC share the same x8 PCIe1.1 slot; according to the PCIe1.1 standard, the maximum *payload* data rate enabled by 8 lanes is 12.8Gbps—slightly above what we actually observe [44]. Hence, the maximum input traffic we can subject our server to is 24.6Gbps. Note that the small number of available NIC slots is an unfortunate consequence of our using a prototype server—as mentioned above, the product server is expected to offer 4–8 slots.

4.2 Exploiting Parallelism

We now summarize the key methods we selected to exploit server parallelism. We illustrate our points with "toy" experiments, for which we use 64B packets and a simple packet-processing scenario where packets are blindly forwarded between pre-determined input and output ports with no header processing or routing lookups.

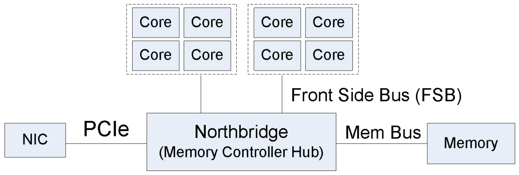

Figure 5: **A traditional shared-bus architecture.**

Multi-core alone is not enough. We first tried building our cluster out of the widely used "shared bus" Xeon servers. Fig. 5 shows a high-level view of this architecture. We see that this earlier server architecture differs from Fig. 4 in two major aspects. First, all communication between the sockets, memory, and I/O devices is routed over the shared front-side bus and "chipset"; in Fig. 4, this communication happens over a mesh of dedicated point-to-point links. Second, the older shared-bus architecture uses a single external memory controller; in Fig. 4, this has been replaced with multiple memory controllers, each integrated within a socket, thus offering a dramatic increase in aggregate memory bandwidth.

The shared-bus Xeon server we used had multiple cores—it had eight 2.4GHz cores, similar to our Nehalem prototype. Yet, for small and even medium-sized packets, its performance fell short of the Nehalem's performance—and well short of our 20Gbps target. In an earlier study [29], we found that the bottleneck was at the shared bus connecting the CPUs to the memory subsystem: Packet processing workloads—as streaming workloads, in general—place a heavy load on memory and I/O, and shared-bus architectures do not provide sufficient bandwidth between the CPUs and these subsystems. Increasing the number of cores does not help—their computational resources are left unexploited, because the cores cannot access the packets fast enough. These results led us to adopt the more recent Nehalem architecture, where parallelism at the CPUs is coupled with parallelism in memory access; this change leads to higher aggregate and per-CPU memory bandwidth, as well as better scaling—no single bus sees its load grow with the aggregate of memory transactions. As we shall see, this improved architecture alone offers a 2–3x performance improvement.

Multi-queue NICs are essential. A packet-processing workload involves moving lots of packets from input to output ports. The question is how should we distribute this workload among the available cores for best effect. We illustrate our approach with the toy scenarios shown in Fig. 6, in which we construct simple forwarding paths (FPs) between pairs of interfaces and cores.

When the receive or transmit queue of a network port is accessed by multiple cores, each core must lock the queue before accessing it—a forbiddingly expensive operation when the port sees millions of packets per second. This leads to our first rule: *that each network queue be accessed by a single core*. We can easily enforce this in Click by associating the polling of each input port and the writing to each output port to a separate thread, and statically assigning threads to cores.

The next question is, how should the processing of a packet be shared among cores? One possibility is the "pipeline" approach, where the packet is handled by multiple cores in a row—one core reads the packet from its receive queue, then passes it to another core for further processing and so on (scenario-(a) in Fig. 6). The alternative is the "parallel" approach, where the packet is read from its receive queue, processed, and written to its transmit queue, all by the same core (scenario-(b) in Fig. 6). For pipelining, we consider both cases where two cores do and don't share the same L3 cache, as this allows us to highlight the performance impact due to the basic overhead of synchronizing cores to transfer the packet from that due to additional cache misses. Comparing the forwarding rates for each case, we see that the parallel approach outperforms the pipelined one in all cases. The overhead just from synchronization across cores can lower performance by as much as 29% (from 1.7 to 1.2Gbps); with additional cache misses, performance drops by 64% (from 1.7 to 0.6Gbps). This leads to our second rule: *that each packet be handled by a single core*.

Hence, we want that each queue and each packet be handled by a single core. The problem is that there are two (very common) cases where it is hard to *simultaneously* enforce both rules. The first case arises when there are many cores and few ports, and a single core cannot by itself handle the processing demands of a port. For instance, consider a 16-core server handling two 10Gbps ports. A single core cannot handle 10Gbps of traffic and hence we'd like to "split" the packet stream across multiple cores. But if a port is tied to a single core, then each packet is necessarily touched by multiple cores (the core that polls in and splits the traffic, and the core that actually processes the packet); this is illustrated in scenario-(c), Fig.6. The second case is when we have "overlapping" paths in which multiple input ports must send traffic to the same output port—scenario-(e) in Fig.6 (compare to scenario-(b)). Overlapping paths arise in all realistic traffic matrices and, once again, tying each port to a single core unavoidably results in multiple cores touching each packet.

Fortunately, both cases can be addressed by exploiting a feature now available in most modern NICs: multiple receive and transmit queues. Multi-queue NICs are used to support virtualization; it turns out that, when coupled with careful scheduling, they also offer a simple, practical solution to our problem [12]. We should note that Bolla and Bruschi also evaluate this approach, albeit in the context of a shared-bus architecture and NICs with multiple 1Gbps interfaces [24]. To leverage multiple NIC queues, we developed a lock-free device driver for 10Gbps multi-queue NICs and extended Click with multi-queue support. Our Click extension allows us to bind polling and sending elements to a particular queue (as opposed to a particular port); this, in turn, allows us to associate each queue to a thread, then statically assign threads to cores in a way that enforces both our rules.

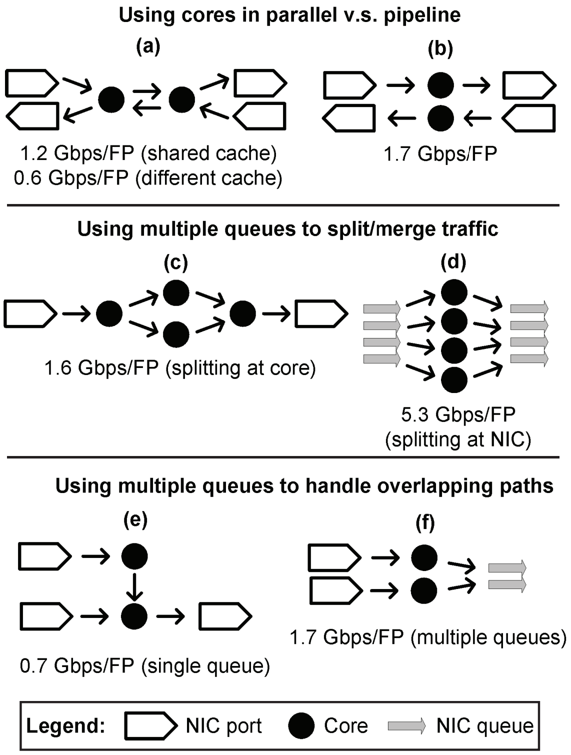
Using cores in parallel v.s. pipeline
(a)
(b)
1.2 Gbps/FP (shared cache)
0.6 Gbps/FP (different cache)
1.7 Gbps/FP
Using multiple queues to split/merge traffic
(c)
(d)
1.6 Gbps/FP (splitting at core)
5.3 Gbps/FP
(splitting at NIC)
Using multiple queues to handle overlapping paths
(e)
(f)
0.7 Gbps/FP (single queue)
1.7 Gbps/FP (multiple queues)
Legend:
NIC port
Core
NIC queue

Figure 6: **Forwarding rates with and without multiple queues.**

In Fig. 6, scenarios-(d) and (f) illustrate how using multiple queues addresses the problematic scenarios in (c) and (e), respectively. In both setups, multi-queue NICs allow us to respect both our "one core per packet" and "one core per interface" rules. We see that the performance impact of leveraging multiple queues is dramatic: *e.g.*, scenario-(d) achieves more than three times higher forwarding rate than scenario-(c); in the case of overlapping paths, we see that, with multiple queues, overlapping paths see forwarding rates similar to those of non-overlapping paths (approximately 1.7Gbps/FP) compared to a performance drop of almost 60% without (0.7Gbps vs. 1.7Gbps/FP).

The next question is whether we always have enough queues to follow this strategy; if a server with m cores has m receive and m transmit queues per port, then the answer is yes. The explanation is straightforward: if each core has its own dedicated receive (transmit) queue at each port, then it can read (write) from any input (output) port without sharing queues or packets with other cores. Multi-queue NICs with 32–64 RX and TX queues already exist, so our solution is feasible today [6]. Moreover, since multiple queues are needed for virtualization, and the number of virtual machines run on a single server is expected to increase with the number of per-server cores, we expect NIC vendors to continue to produce multi-queue NICs where the number of queues follows the number of per-server cores. Hence, unless stated otherwise, from here on we use multi-queue NICs as described above.

Polling configuration	Rate (Gbps)
No batching ($k_p = k_n = 1$)	1.46
Poll-driven batching ($k_p = 32, k_n = 1$)	4.97
Poll-driven and NIC-driven batching ($k_p = 32, k_n = 16$)	9.77

Table 1: **Forwarding rates achieved with different polling configurations. k_p=32 is the default Click maximum. We stop at k_n=16 because the maximum PCIe packet size is 256B; a packet descriptor is 16B, hence, we can pack at most 16 descriptors in a single PCIe transaction.**

"Batch" processing is essential. Forwarding a packet involves a certain amount of per-packet book-keeping overhead—reading and updating socket buffer descriptors and the data structures (ring buffers) that point to them. This overhead can be reduced by "bulk" processing descriptors, *i.e.*, amortizing book-keeping operations by incurring them once every k packets. The standard approach is to drive such batching from the application : specifically, Click can receive up to k_p packets per poll operation—we call this "poll-driven" batching. To this, we added "NIC-driven" batching: we extended our NIC driver to relay packet descriptors to/from the NIC only in batches of k_n packets. This results in fewer (but larger) transactions on the PCIe and I/O buses and complements poll-driven batching by ensuring that at least k_n packets are available to be polled in at a time.

We measure our server's maximum forwarding rate using all 8 cores and various polling configurations. Table 1 shows the results: Poll-driven batching offers a 3-fold performance improvement relative to no batching, while adding NIC-driven batching improves performance by an additional factor of 2. Hence, unless stated otherwise, from here on we configure our servers with batching parameters $k_p = 32$ and $k_n = 16$.

Batching increases latency (since the NIC waits for k_n packets to arrive before initiating a PCIe transaction) and jitter (since different packets may have to wait at the NIC for different periods of time). On the other hand, batching is necessary only at high speeds, where packet inter-arrival times are small (on the order of nanoseconds), hence, the extra latency and jitter are expected to be accordingly small. At lower packet rates, increased latency can be alleviated by using a timeout to limit the amount of time a packet can wait to be "batched" (we have yet to implement this feature in our driver).

NUMA-aware data placement is not. We initially expected careful data placement to be essential in maximizing performance since the Nehalem is a NUMA architecture. This expectation was strengthened when we found that Linux does not always place data in an ideal way: even though the packets themselves are ideally placed (each one closest to the core that reads it from the NIC), socket-buffer descriptors are always placed in one particular memory (socket-0, the one that belongs to CPU-0), independently of which core is processing the packet.

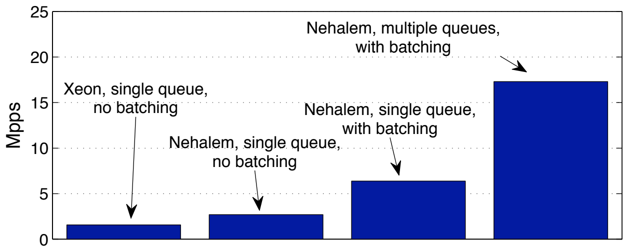

Figure 7: **Aggregate impact on forwarding rate of new server architecture, multiple queues, and batching.**

Surprisingly, we found that, at least for our workloads, careful data placement makes no difference in performance. We demonstrate this through a simple setup: we disable the cores on socket-1 and measure the maximum forwarding rate achieved by the 4 cores on socket-0; in this case, both packets and socket-buffer descriptors are ideally placed in the memory of socket-0—we record a forwarding rate of 6.3Gbps. We then repeat the experiment but this time disable the cores on socket-0 and use only the 4 cores in socket-1; in this case, the packets are placed in the memory of socket-1, while the descriptors are placed in the "remote" memory of socket 0. In this latter case, we find that approximately 23% of memory accesses are to remote memory (our tools breakdown memory accesses as local *vs.* remote), nonetheless, we get a forwarding rate of 6.3Gbps. Hence, we conclude that custom data placement is not critical. This is not to suggest that careful data placement is never required—just that, for our particular (fairly memory-intensive) workload and server architecture, it isn't.

Putting it all together. In summary, we found that the performance potential of multi-core servers is best exploited, when parallelism at the CPUs is accompanied by parallelism in memory access (through dedicated memory controllers and buses) and NICs (through multiple queues), and if the lower levels of the software stack are built to leverage this potential (through batching). To this end, we took an existing 10Gbps NIC driver and added support for (1) polling/writing to multiple queues and (2) configurable batching of socket-buffer descriptor operations.

We now look at the cumulative effect of our design lessons. We record the forwarding rate achieved by our server using all eight cores, four 10Gbps ports, 64B packets, under the same simple forwarding as in the toy scenarios (*i.e.*, packets are forwarded between pre-determined input and output ports) and a uniform any-to-any traffic pattern (*i.e.*, traffic from each input is uniformly split across all output ports). We repeat this experiment four times: (1) using an 8-core Xeon server without any of our changes (*i.e.*, no multi-queue NICs and no batching), (2) our Nehalem server without any of our changes, (3) our Nehalem server with multiple queues but no batching, and (4) our Nehalem server with both multiple queues and batching. Fig. 7 shows the results: we see that our modifications lead to a 6.7-fold (670%) improvement relative to the same server without our modifications and an 11-fold improvement relative to the shared-bus Xeon.

Thus, we find that our modest design changes significantly impact performance. We arrived at these through a more careful (than usual) understanding of the underlying hardware—its raw resource capability, potential bottlenecks, and contention points. While some "awareness" of hardware is always useful in optimizing performance, we found that, with greater parallelism in server hardware, the performance impact of this awareness can be quite dramatic. At the same time, we note that our modifications actually have little-to-no impact on Click's programming model. Finally, we note that, although we focus on packet forwarding, we expect the above findings would apply to the more general class of streaming applications (*e.g.*, continuous query processors, stock trading, or video streaming).

5 Evaluation: Server Parallelism

Having established how we parallelize packet-processing within a server, we now evaluate the resulting server performance for different workloads. After describing our test workloads (§5.1), we start with black-box testing (§5.2), then analyze the observed performance (§5.3). For clarity, in this section we evaluate performance in the context of one server and one workload at a time; §6 then looks at server performance in a complete VLB-cluster router.

5.1 Workloads

At a high level, a packet-processing workload can be characterized by (1) the distribution of *packet sizes*, and (2) the *application*, *i.e.*, the type of processing required per packet. As far as packet size is concerned, we consider synthetic workloads, where every packet has a fixed size of P bytes, as well as a trace-driven workload generated from the "Abilene-I" packet trace collected on the Abilene network [10]. As far as the application is concerned, we consider the following three:

1. *Minimal forwarding:* In this application, traffic arriving at port i is just forwarded to port j—there is no routing-table lookup nor any other form of packet processing. This simple test is valuable for several reasons. First, it stresses our ability to rapidly stream large volumes of cache-unfriendly data through the system, the key requirement that makes packet processing challenging. Second, it exercises the minimal subset of operations that *any* packet-processing (for that matter, any streaming) application incurs and consequently offers an upper bound on the achievable performance for *all* such applications. Finally, minimal forwarding is precisely the type of processing performed by VLB nodes when they act as intermediate or destination nodes.

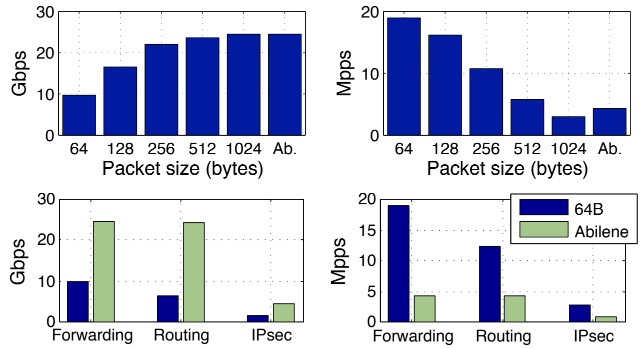

Figure 8: **Forwarding rate for different workloads. Top: as a function of different packet-size distributions, when the server performs minimal forwarding. Bottom: as a function of different packet-processing applications, for 64B packets and the Abilene trace. "Ab." refers to the Abilene trace.**

2. *IP routing:* We implement full IP routing including checksum calculations, updating headers, and performing a longest-prefix-match lookup of the destination address in a routing table. For this latter, we use the Click distribution's implementation of the D-lookup algorithm [34] and, in keeping with recent reports, a routing-table size of 256K entries. For synthetic input traffic, we generate packets with random destination addresses so as to stress cache locality for IP lookup operations.

3. *IPsec packet encryption:* In this application, every packet is encrypted using AES-128 encryption, as is typical in VPNs.

Our selection represents commonly deployed packet-processing applications that are fairly diverse in their computational needs. For example, minimal forwarding stresses memory and I/O; IP routing additionally references large data structures; encryption is CPU-intensive.

Given this setup, our primary performance metric is the maximum attainable loss-free forwarding rate, reported in terms of either bits per second (bps) or packets per second (pps).

5.2 Black-Box System Performance

First, we measure the maximum loss-free forwarding rate our server can sustain when running the minimal-forwarding application, given (1) input traffic of fixed-size packets, repeated for different packet sizes, and (2) the Abilene trace. The resulting rate is shown in Fig. 8 (top), in both bps and pps. We see that, given larger packets or the Abilene trace, the server sustains 24.6Gbps; this is the maximum input traffic we can generate, meaning that performance, in this case, is limited by the number of NICs our server can fit—we do not hit any bottleneck inside the server. In contrast, given 64B packets, the server saturates at 9.7Gbps or 18.96Mpps.

We observe a similar relationship between packet size and performance for all applications. Hence, from here on, we focus on (1) fixed-size packets of minimum length ($P = 64$B) and (2) the Abilene trace. Fig. 8 (bottom) shows server performance for our three different applications. We see that performance drops as the per-packet processing demands increase: for IP routing, the server saturates at 24.6Gbps given the Abilene trace and 6.35Gbps given 64B packets; for IPSec, the rates are even lower—4.45Gbps given Abilene, 1.4Gbps given 64B packets. We should note that, compared to the state of the art, our IPsec rates are still commendable—routers typically use additional IPsec accelerators to scale to 2.5Gbps [4]–10Gbps [2].

We now look inside our server to understand these black-box performance results.

5.3 Deconstructing System Performance

Our approach is to probe the limits of the server's components with the aim of understanding what bottleneck(s) currently limit performance and how performance may be expected to scale with future servers. Our methodology is as follows. We consider each of the major system components: (1) CPUs, (2) memory buses, (3) the socket-I/O links, (4) the inter-socket link, and (5) the PCIe buses connecting the NICs to the I/O hub (see Fig. 4).[3] For each component, we estimate an *upper bound* on the per-packet load that the component can accommodate, as a function of the input packet rate. Then we measure the per-packet load on the component under increasing input packet rates, given different workloads. Comparing the actual loads to our upper bounds reveals which components are under greatest stress and, hence, likely to be bottlenecks.

We directly measure the actual loads on the system buses using our tools. Computing the per-packet CPU load requires more attention, because Click operates in polling mode, hence, the CPUs are always 100% utilized. To compute the "true" per-packet CPU load, we need to factor out the CPU cycles consumed by *empty* polls—*i.e.*, cycles where the CPU polls for packets to process but none are available in memory. We do this by measuring the number of cycles consumed by an empty poll (c_e) and the number of empty polls measured per second (E_r) for each input packet rate r; deducting $c_e \times E_r$ from the server's total number of cycles per second gives us the number of cycles per second consumed by packet processing for each input packet rate r.

We consider two approaches for estimating upper bounds on the per-packet loads achievable by our server's components. The first one is based on the nominal rated capacity of each component. For example, our server has eight 2.8GHz cores and, hence, we estimate an upper bound of $\frac{8 \times 2.8 \times 10^9}{r}$ cycles/packet given an input packet rate r. For certain components (*e.g.*, the memory buses), actually achieving this nominal capacity is known to be difficult. Hence, we also consider a second, empirical approach that uses benchmarks, specifically designed to impose a high load on the target component. For example, to estimate an upper bound on the per-packet load achievable by the memory buses, we wrote a simple "stream" benchmark that writes a constant value to random locations in a very large array, and measured the resulting load, M_{strm}, on the memory buses, in bytes/second. We then estimate the maximum per-packet load achievable by the memory buses as $\frac{M_{strm}}{r}$ bytes/packet, given an input packet rate r.

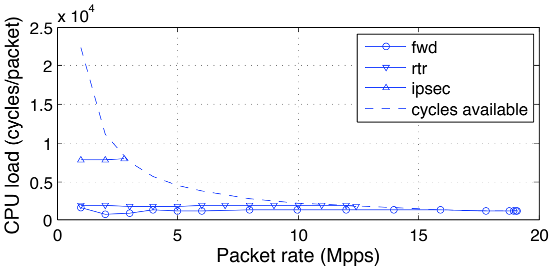

Figure 9: **CPU load (in cycles/packet) as a function of incoming traffic rate (in packets/sec) for different packet-processing applications. In the legend, "fwd" corresponds to minimal forwarding and "rtr" to IP routing, while "cycles available" corresponds to the nominal capacity of the CPUs.**

Table 2 summarizes the nominal and empirical upper bounds we derived for each system component. Figs. 9 and 10 plot both of these upper bounds, as well as the per-packet loads measured on each system component, for each of our three applications and 64B packets. We draw the following conclusions from these results:

1) Bottlenecks. We see that, for all three applications, the measured CPU load approaches the nominal upper bound, indicating that the CPUs are the bottleneck in all three cases[4] (Fig. 9). The next question is whether the CPUs are efficient in their packet processing, *i.e.*, whether they spend their cycles doing useful work, as opposed to, for example, waiting for memory accesses to complete. We answer this question by breaking down CPU load (cycles per packet) into instructions per packet and cycles per instruction (CPI) for each application. These are listed in Table 3. Nehalem processors can retire up to 4 instructions/cycle leading to a minimum

[3] We do not directly consider L3 caches, since any increase in cache miss rates appears as load on the memory buses, which we do consider.

[4] We can conclude this because the cycles/packet remains constant under increasing packet rates. If, instead, the cycles/packet were growing with input packet rate, then we would have to consider the possibility that the true problem is at the memory system. In that case, the problem would be that higher packet rates stress the memory system leading to higher memory access times, which in turn lead to higher CPU cycles/packet since the CPUs spend more time waiting for memory accesses to return.

Component(s)	Nominal capacity	Benchmark for empirical upper-bound
CPUs	8 × 2.8 GHz (#cores×cpu-speed)	None
Memory	410 Gbps (#mem-buses×bus-capacity)	262 Gbps (stream with random access)
Inter-socket link	200 Gbps [19]	144.34 Gbps (stream)
I/O-socket links	2 × 200 Gbps [19]	117 Gbps (min. forwarding with 1024B packets)
PCIe buses (v1.1)	64 Gbps (2 NICs × 8 lanes× 2 Gbps per direction) [44]	50.8 Gbps (min. forwarding with 1024B packets)

Table 2: **Upper bounds on the capacity of system components based on nominal ratings and empirical benchmarks.**

Application	instructions/packet	cycles/instruction
Minimal forwarding	1,033	1.19
IP routing	1,512	1.23
IPsec	14,221	0.55

Table 3: **Instructions-per-packet (IPP) and cycles-per-instruction (CPI) for 64B packet workloads.**

CPI of 0.25 [19]. Discussion with CPU architects reveals that, as a rule of thumb, a CPI of 0.4–0.7, for CPU-intensive workloads, and 1.0–2.0, for memory-intensive workloads, is regarded as efficient CPU usage. We thus conclude that our CPUs are efficiently used; moreover, in the case of minimal forwarding, the small number of instructions per packet shows that Click's software architecture is efficient. In other words, a poor software architecture is not the problem; performance truly is limited by a lack of CPU cycles.

We note that having the CPUs as the bottleneck is not undesirable, since this (finally) aligns the performance needs of router workloads with the vast majority of PC applications. Hence, software routers stand to benefit from the expectation that the number of cores will scale with Moore's law [33].

2) Small *vs.* large packets. We compared the per-packet load imposed on the system by 1024B-packet workloads to that imposed by 64B-packet workloads (we omit the graph for brevity). A 1024B packet is 16 times larger than a 64B one, so, initially, we expected the load imposed by each 1024B packet on each system bus to be 16 times larger than the load imposed by a 64B packet. Yet, we found that it is only 6, 11, and 1.6 times larger, respectively, for the memory buses, I/O, and CPU. This means that the per-*byte* load is higher for smaller packets. In retrospect, this makes sense due to the book-keeping performed for each packet, which is independent of packet size; for larger packets, book-keeping overhead is amortized across more bytes.

3) Non-bottlenecks. We see that the per-packet loads on the memory and I/O buses are well below their empirically derived upper bounds, indicating that these traditional problem areas for packet processing are no longer the primary performance limiters. Likewise, a traditional concern regarding multi-socket architectures is the scalability of the inter-socket interconnects; however, we see that these links are not heavily loaded for our workloads.

4) Expected scaling. Finally, we see that, for all three appli-

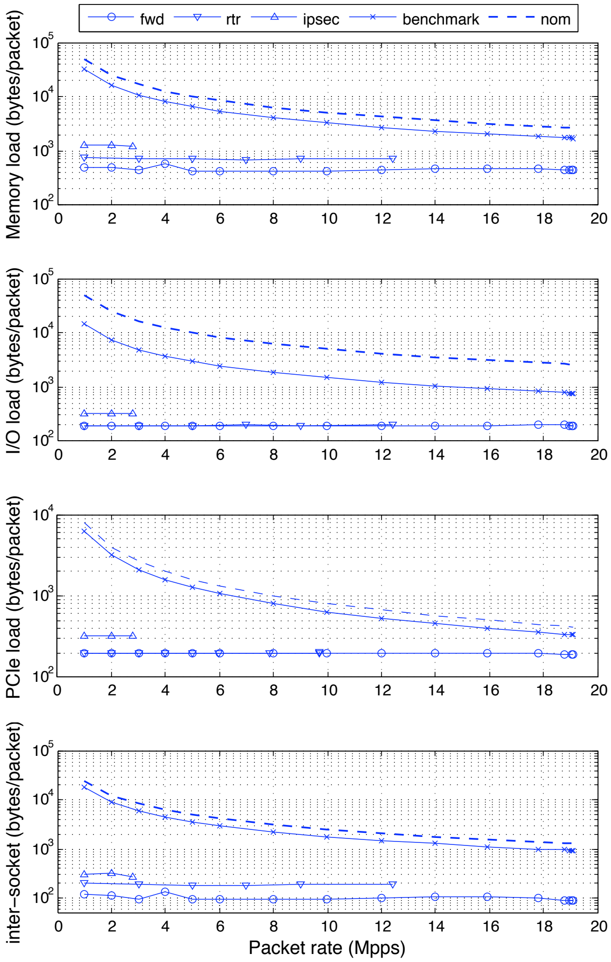

Figure 10: **Load on system buses (in bytes/packet) as a function of the incoming traffic rate (in packets/sec). From top to bottom: memory buses, socket-I/O links, PCIe buses, and inter-socket links.**

cations and all packet-size distributions, the per-packet load on the system is constant with increasing input packet rate. This allows us to extrapolate in a straightforward manner how performance, for these particular applications, is expected to scale with next-generation servers. As an example, we consider the expected follow-up to our server, which has 4 sockets and 8 cores per socket, thus offering a $4x$, $2x$ and $2x$ increase in total CPU, memory, and I/O resources, respectively (Nehalem is designed to scale up to 8 cores [33]). Adjusting the upper bounds in Figs. 9–10 accordingly and extrapolating where the observed loads would intersect with the new upper bounds, we project performance rates of 38.8, 19.9, and 5.8Gbps for minimal forwarding, IP routing, and IPsec, respectively, given 64B packets, and find that the CPU remains the bottleneck. Similarly, we can estimate the performance we might have obtained given the Abilene trace, had we not been limited to just two NIC slots: ignoring the PCIe bus and assuming the socket-I/O bus can reach 80% of its nominal capacity, we estimate a performance of 70Gbps for the minimal-forwarding application given the Abilene trace. These are, of course, only projections, and we intend to validate them when possible.

In summary, we found that current servers achieve commendable performance given the realistic Abilene workloads (minimal forwarding: 24.6Gbps, IP routing: 24.6Gbps), but fare worse given the worst-case 64B-packet workloads (minimal forwarding: 9.7Gbps, IP routing: 6.35Gbps). We showed that the CPU is the bottleneck, but estimated that next-generation servers are expected to offer a 4-fold performance improvement. In the next section, we look at server performance in a complete cluster router.

6 The RB4 Parallel Router

We built a prototype parallel router based on the design and performance lessons presented so far. Our router—the RB4—consists of 4 Nehalem servers, interconnected through a full-mesh topology with Direct-VLB routing (§3). Each server is assigned a single 10Gbps external line.

6.1 Implementation

We start with a straightforward implementation of the algorithms described earlier, then add certain modifications to Direct VLB, aimed at reducing the load on the CPUs (since we identified these as our bottleneck) and avoiding reordering (the issue we deferred from §3). We discuss each in turn.

Minimizing packet processing. In Direct VLB, each packet is handled by 2 or 3 nodes (2, when it is directly routed from its input to its output node, 3, when it is routed via an intermediate node). The straightforward implementation would be to have, at each node, the CPU process the packet's header and determine where to send it next, which would result in each packet's header being processed by a CPU 2 or 3 times. Instead, in RB4, each packet's header is processed by a CPU *only once*, at its input node; subsequent nodes simply move the packet from a receive to a transmit queue. To achieve this, we leverage a NIC feature that assigns packets to receive queues based on their MAC addresses.

More specifically: When a packet arrives at its input node, one of the node's CPUs processes the packet's headers and encodes the identity of the output node in the packet's MAC address. At each subsequent node, the packet is stored in a receive queue based on its MAC address; hence, by looking at the receive queue where the packet is stored, a CPU can deduce the packet's MAC address and, from that, the packet's output node. In this way, the CPU that handles the packet at subsequent nodes can determine where to send it without actually reading its headers. We should clarify that this particular implementation works only if each "internal" port (each port that interconnects two cluster servers) has as many receive queues as there are external ports—hence, with current NICs, it would not be applicable to a router with more than 64 or so external ports.

Avoiding reordering. In a VLB cluster, two incoming packets can be reordered because they take different paths within the same server (due to multiple cores) or across the cluster (due to load balancing). One approach to avoiding reordering relies on perfectly synchronized clocks and deterministic per-packet processing latency [37]; we reject this, because it requires custom operating systems and hardware. Another option would be to tag incoming packets with sequence numbers and re-sequence them at the output node; this is an option we would pursue, if the CPUs were not our bottleneck. Instead, we pursue an alternative approach that mostly avoids, but does not completely eliminate reordering.

We try to avoid reordering *within each TCP or UDP flow*—after all, the main reason for avoiding reordering is that it can affect TCP or streaming performance. First, same-flow packets arriving at a server are assigned to the same receive queue. Second, a set of same-flow packets arriving at the cluster within δ msec from one another are sent, whenever possible, through the same intermediate node—this is akin to the Flare load-balancing scheme [35]. When a burst of same-flow packets (a "flowlet" in Flare terminology) does not "fit" in one path (*i.e.*, sending the whole flowlet to the same intermediate node would overload the corresponding link), then the flowlet is load-balanced at the packet level as in classic VLB. We found that, in practice, $\delta = 100$msec (a number well above the per-packet latency introduced by the cluster) works well, *i.e.*, allows most flowlets to be sent through one path, thereby avoiding reordering (§6.2).

6.2 Performance

Forwarding performance. Given a workload of 64B packets, we measure RB4's routing performance at 12Gbps, *i.e.*, each server supports an external line rate of 3Gbps. This number is in keeping with our expectations: VLB theory tells us that, in a 4-node Direct-VLB cluster of external rate R, each node must process packets at a rate between $2R$ (when all packets are directly routed) and $3R$ (§3.2). Given a 64B packet workload, RB4 routes all packets directly (because the traffic rate between any two nodes is never enough to saturate the link between them), hence, each node must process packets at rate $2R$. Moreover, we know that, given 64B packets, a single server achieves a maximum processing rate of 9.7Gbps when running minimal forwarding and 6.35Gbps when running IP routing (§5). In RB4, each server performs both IP routing (for packets entering the cluster at that server) and minimal forwarding (for packets exiting the cluster at that server, or being load-balanced during the second VLB phase). Hence, we expected RB4 performance to lie between $4 \times \frac{6.35}{2} = 12.7$ and $4 \times \frac{9.7}{2} = 19.4$Gbps. The reason for RB4's somewhat lower performance is due to the extra overhead caused by the reordering-avoidance algorithm (recall that our bottleneck is the CPU, and reordering avoidance requires it to maintain per-flow counters and packet-arrival times, as well as keep track of link utilization to avoid overloading).

Given the Abilene workload, we measure RB4's routing performance at 35Gbps, which, again, is in keeping with what we expected: A single server (running either minimal forwarding or IP routing) can process the Abilene workload at 24.6Gbps (§5), hence, we expected RB4 to process the same workload at a rate between $4 \times \frac{24.6}{3} = 33$ and $4 \times \frac{24.6}{2} = 49$Gbps (the latter for a perfectly uniform traffic matrix). At the same time, the performance of our prototype is constrained by the limit of approximately 12Gbps that a single (dual port) NIC can sustain, as described in §4.1. At 35Gbps, we are close to this per-NIC limit: each NIC that handles an external line sustains approximately 8.75Gbps of traffic on the port connected to the external line plus approximately 3Gbps on the second "internal" port. So, RB4's performance is within the expected range, but, unfortunately, the limited number of PCIe slots on our prototype server (because of which we face the per-NIC limit) prevents us from precisely quantifying where—in the range between $2R$ and $3R$—VLB implementation overhead lies. In future work, we plan to upgrade our server motherboards to fully explore this overhead.

Reordering. To measure the amount of reordering introduced by RB4, we replay the Abilene trace, forcing the entire trace to flow between a single input and output port—this generated more traffic than could fit in any single path between the two nodes, causing load-balancing to kick in, hence, creating opportunity for reordering. We measure reordering as the fraction of same-flow packet sequences that were reordered within their TCP/UDP flow; for instance, if a TCP flow consists of 5 packets that enter the cluster in sequence $\langle p_1, p_2, p_3, p_4, p_5 \rangle$ and exit the cluster in sequence $\langle p_1, p_4, p_2, p_3, p_5 \rangle$, we count one reordered sequence. With this metric, we observe 0.15% reordering when using our reordering-avoidance extension and 5.5% reordering when using Direct VLB without our extension.

Latency. We estimate packet latency indirectly: a packet's traversal through a single server involves two back-and-forth DMA transfers between the NIC and memory (one for the packet and one for its descriptor) plus processing by the CPUs. In addition, NIC-driven batching means that a packet may wait for 16 packets before transmission. We estimate a DMA transfer for a 64B packet at 2.56μsecs based on our DMA engine speed of 400MHz and published reports [50]. From Table 3, routing a 64B packet takes 2425 cycles or 0.8μsecs and, hence, batching can add up to 12.8μsecs. Thus, we estimate a per-server packet latency of 24μs $(4 \times 2.56 + 12.8 + 0.8)$. Traversal through RB4 includes 2–3 hops; hence we estimate RB4's latency as $47.6 - 66.4\mu$s. As an additional point of reference, [42] reports a packet-processing latency of 26.3μs for a Cisco 6500 Series router.

7 Related Work

Using general-purpose elements to build programmable routers is not a new idea. The original NSFNET used computers running Berkeley UNIX interconnected with a 4Mbps IBM token ring [26]. MGR, an early high-speed router, used custom forwarding-engine cards interconnected through a switched bus, yet the processor on these cards was a general-purpose one [43]. This combination of custom cards/interconnect with general-purpose processors was also used in commercial routers, until router manufacturers transitioned to ASIC-based forwarding engines for higher performance. More recently, single-server software routers have emerged as low-cost solutions for low-speed (1–5Gbps) environments [16]. Our contribution lies in detailing how multiple servers can be clustered to achieve greater scalability and studying how modern servers can be exploited to this end.

Several efforts have sought to reconcile performance and programmability using network processors (NPs) [45]. Most recently, Turner *et al.* proposed a "supercharged" Planetlab Platform, which uses IXP NPs (for the data plane) and general-purpose servers (for the control plane), interconnected by an Ethernet switch; they achieve forwarding rates of 5Gbps for 130B packets [46]. We focus, instead, on using general-purpose servers even on the data plane, and our results indicate these offer competitive performance.

Click [25, 38] and Scout [45] explored how to architect router software so as to enable easy programmability and extensibility; SMP Click [25] extended the early Click architecture to better exploit multiprocessor PCs. We also started with Click, extended it to exploit new server technologies, studied the performance benefits of these technologies, and, finally, applied Click to building a cluster-based router.

Bianco *et al.* measured the routing performance of a single-core server equipped with a PCI-X (rather than PCIe)

I/O bus; they found that the bottlenecks were the (single) CPU and the PCI-X bus [23]. More recent work studied the performance of multicore shared-bus Xeon servers in the context of TCP termination [48] and virtual routers [31], and we, also, studied the packet-processing capabilities of that architecture [29]; these studies report that the bottleneck lies in the shared bus (the "front-side bus" or FSB) connecting the CPUs to the memory subsystem. In contrast, RouteBricks relies on a newer server architecture; we showed that this enables a $2-3x$ performance improvement, and that the packet-processing bottleneck now lies at the CPUs.

Finally, our work extends an earlier workshop paper [22], where we made the case for scaling software routers and proposed a cluster-based approach [22]; in this paper, we presented a detailed design, implementation, and evaluation for that approach.

8 Discussion

We evaluated the feasibility of our router architecture from the standpoint of performance—the traditional Achilles' heel of software routers. However, its ultimate feasibility depends on additional, equally important issues, such as space and power consumption, where any solution based on general-purpose server components faces tough competition from solutions using custom hardware. On the other hand, server-based solutions enable programmability and extensibility. So, to compare a server-based, programmable router to a specialized, non-programmable one, we would have to quantify the benefits of programmability—*e.g.*, would a 20% increase in power consumption be worth the ability to rapidly upgrade network functionality, say, to protect against a new type of worm? We do not attempt such a comparison here—it merits to be the topic of a research project in itself. We only discuss space and power consumption, as well as cost, briefly, and offer a few data points on how current routers fare, as relevant points of reference.

Form factor. Router form factor is typically a question of port density. Off the shelf, the RB4 would be a 40Gbps router (assuming we can close our remaining performance gap) that occupies 4U, which is not unreasonable. However, scaling RB4 up would be problematic, especially if we introduce additional servers to cope with the limited per-server fanout. One avenue is to grow the per-server fanout; however, doing so by adding external NIC slots would lead to larger servers. Instead, we can integrate Ethernet controllers directly on the motherboard (commonly done for laptops, requiring only hardware reconfiguration). The question is whether we can integrate many such controllers and still satisfy concerns over cooling and chipset area. We estimate that a regular 400mm motherboard could accommodate 16 controllers to drive $2 \times$ 10Gbps and $30 \times$ 1Gbps interfaces for a reasonable $+48$W. With this, we could directly connect 30–40 servers. Thus, 1U servers, each handling one 10Gbps external line, would result in a 300–400Gbps router that occupies 30U. In addition, form factor will benefit from server advances; *e.g.*, we estimated that the 4-socket Nehalem would offer a $4x$ performance improvement (§5.3) and, hence, for the same performance, we can expect to reduce form factor by $4x$. The above is, of course, just a back-of-the-envelope estimate that requires evaluation before we draw final conclusions. For reference, we note that the Cisco 7600 Series [3] offer up to 360Gbps in a 21U form-factor.

Power. Based on our server's nominal power rating, the RB4 consumes 2.6KW. As a point of reference, the nominal power rating of a popular mid-range router loaded for 40Gbps is 1.6KW [3]—about 60% lower. One approach to reducing power consumption would be to slow-down, or put to sleep, system components that are not stressed by router workloads, using commonly available low-power modes for different subsystems (memory, serial links, floating-point units).

Cost/Price. With respect to the price/cost: our RB4 prototype cost us $\$14,500$; for reference, the quoted price for a 40Gbps Cisco 7603 router was $\$70,000$. Again, this should not be viewed as a direct comparison, since the former represents raw costs, while the latter is a product price.

Programmability. Our high-level goal was to achieve both high performance and ease of programming. Hence, we started with Click—which offers an elegant programming framework for routing applications [38]—and sought to maximize performance without compromising Click's programming model. Our design maintained Click's modularity and extensibility; our only intervention was to enforce a specific element-to-core allocation. As a result, our router is not just programmable in the literal sense (*i.e.*, one can update its functionality), it also offers ease of programmability. Case in point: beyond our 10G NIC driver, the RB4 implementation required us to write only two new Click "elements"; the effort to develop a stable NIC driver far exceeded the effort to write our two Click elements and tie them with other pre-existing elements in a Click configuration.

As a next step, future research must demonstrate what new kinds of packet processing RouteBricks enables and how these affect performance. Since we have identified CPU processing as the bottleneck of our design, we expect performance to vary significantly as a function of the application, as showcased by the IP-routing and IPsec experiments (§5). Hence, the key challenge will be to identify the right API (perhaps the right Click extension), which will allow the programmer not only to easily add new, non-traditional functionality, but also to easily predict and control the resulting performance implications.

9 Conclusions

We looked to scale software routers as a means of moving from a network of special-purpose hardware routers to one of general-purpose infrastructure. We proposed a parallel router architecture that parallelizes routing functionality both across and within servers. The goal we set was high: essentially to match the performance of even high-end routers (with line rates of 10Gbps and 10s or 100s of ports). We cannot claim that our results, based on today's servers, make a slam-dunk case for getting there. That said, they do show that we are a whole lot closer than common expectations for software routers would have led us to believe: we can comfortably build software routers with multiple (about 8–9) 1Gbps ports per server, which we can scale to 10s or 100s of ports by grouping multiple servers; we come very close to achieving a line rate of 10Gbps and, importantly, show that emerging servers promise to close the remaining gap to 10Gbps, possibly offering up to 40Gbps. The broad implications of this are twofold: one is that software routers could play a far more significant role than previously believed; the more ambitious extrapolation is that a very different industry structure and way of building networks might actually be within not-so-distant reach.

10 Acknowledgments

We would like to thank the following colleagues for their insightful comments: Tim Brecht, Brighten Godfrey, Ming Iu, Eddie Kohler, Petros Maniatis, Laurent Mathy, Sergiu Nedevschi, Simon Schubert, and Cristian Zamfir.

References

[1] Astaro: Security Gateway. `http://www.astaro.com`.

[2] Cavium Octeon Multi-Core Processor Family. `http://caviumnetworks.com/OCTEON_MIPS64.html`.

[3] Cisco 7600 Series Routers. `http://cisco.com/en/US/products/hw/routers/ps368/index.html`.

[4] Cisco 7600 VPN Port Adapter. `http://cisco.com/en/US/products/ps6917/index.html`.

[5] Cisco Carrier Routing System. `http://cisco.com/en/US/products/ps5763/index.html`.

[6] Intel 10 Gigabit XF SR Server Adapters. `http://intel.com/Products/Server/Adapters/10-GbE-XFSR-Adapters/10-GbE-XFSR-Adapters-overview.htm`.

[7] Intel VTune Performance Analyzer. `http://software.intel.com/en-us/intel-vtune/`.

[8] Narus: Real-Time Traffic Intelligence. `http://narus.com`.

[9] NetFPGA: A Line-rate, Flexible Platform for Research and Classroom Experimentation. `http://netfpga.org`.

[10] NLANR: Internet Measurement and Analysis. `http://moat.nlanr.net`.

[11] Principles and Practices of Interconnection Networks, Chapter 6. William J. Dally and Brian Towles, Morgan Kaufmann, 2004.

[12] Receive-Side Scaling Enhancements in Windows Server 2008. `http://www.microsoft.com/whdc/device/network/NDIS_RSS.mspx`.

[13] Riverbed: Application Acceleration. `http://www.riverbed.com`.

[14] Sourcefire: Network Security. `http://www.sourcefire.com`.

[15] Symantec: Data Loss Protection. `http://www.vontu.com`.

[16] Vyatta Series 2500. `http://vyatta.com/downloads/datasheets/vyatta_2500_datasheet.pdf`.

[17] Cisco Opening Up IOS. `http://www.networkworld.com/news/2007/121207-cisco-ios.html`, December 2007.

[18] Juniper Press Release: Open IP Solution Program. `http://www.juniper.net/company/presscenter/pr/2007/pr-071210.html`, July 2007.

[19] Next Generation Intel Microarchitecture (Nehalem). `http://intel.com/pressroom/archive/reference/whitepaper_Nehalem.pdf`, 2008.

[20] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity, Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM Conference*, Seattle, WA, USA, August 2008.

[21] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet Denial-of-Service with Capabilities. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, Cambridge, MA, USA, November 2003.

[22] K. Argyraki, S. Baset, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, E. Kohler, M. Manesh, S. Nedevschi, and S. Ratnasamy. Can Software Routers Scale? In *Proceedings of the ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of TOmorrow (PRESTO)*, Seattle, WA, USA, August 2008.

[23] A. Bianco, R. Birke, D. Bolognesi, J. M. Finochietto, G. Galante, and M. Mellia. Click vs. Linux. In *Proceedings of the IEEE Workshop on High Performance Switching and Routing (HPSR)*, Hong Kong, May 2005.

[24] R. Bolla and R. Bruschi. PC-based Software Routers: High Performance and Application Service Support. In *Proceedings of the ACM SIGCOMM Wokshop on Programmable Routers for Extensible Services of TOmorrow (PRESTO)*, Seattle, WA, USA, August 2008.

[25] B. Chen and R. Morris. Flexible Control of Parallelism in a Multiprocesor PC Router. In *USENIX Technical Conference*, 2001.

[26] B. Chinoy and H.-W. Braun. The National Science Foundation Network. Technical Report GA-21029, SDSC Applied Network Research group, 1992.

[27] D. Comer. *Network System Design using Network Processors*. Prentice Hall, 2004.

[28] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.

[29] N. Egi, M. Dobrescu, J. Du, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, L. Mathy, and S. Ratnasamy. Understanding the Packet Processing Capabilities of Multi-Core Servers. Technical Report LABOS-REPORT-2009-001, EPFL, Switzerland, February 2009.

[30] N. Egi, A. Greenhalgh, mark Handley, M. Hoerdt, F. Huici, and L. Mathy. Fairness Issues in Software Virtual Routers. In *Proceedings of the ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of TOmorrow (PRESTO)*, Seattle, WA, USA, August 2008.

[31] N. Egi, A. Greenhalgh, mark Handley, M. Hoerdt, F. Huici, and L. Mathy. Towards High Performance Virtual Routers on Commodity Hardware. In *Proceedings of the ACM International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, Madrid, Spain, December 2008.

[32] R. Ennals, R. Sharp, and A. Mycroft. Task Partitioning for Multi-Core Network Processors. In *Proceedings of the IEEE International Conference on Computer Communications (ICCC)*, Mauritius, April 2005.

[33] P. P. Gelsinger. Intel Architecture Press Briefing. `http://download.intel.com/pressroom/archive/reference/Gelsinger_briefing_0308.pdf`, March 2008.

[34] P. Gupta, S. Lin, and N. McKeown. Routing Lookups in Hardware at Memory Access Speeds. In *Proceedings of the IEEE INFOCOM Conference*, San Francisco, CA, USA, March 1998.

[35] S. Kandula, D. Katabi, S. Sinha, and A. Berger. Flare: Responsive Load Balancing Without Packet Reordering. *ACM Computer Communications Review (CCR)*, 37(2), April 2007.

[36] D. Katabi, M. Handley, and C. Rohrs. Internet Congestion Control for High Bandwidth-Delay Product Networks. In *Proceedings of the ACM SIGCOMM Conference*, Pittsburgh, PA, USA, August 2002.

[37] I. Keslassy, S.-T. Chuang, K. Yu, D. Miller, M. Horowitz, O. Solgaard, and N. McKeown. Scaling Internet Routers Using Optics. In *Proceedings of the ACM SIGCOMM Conference*, Karlsruhe, Germany, August 2003.

[38] E. Kohler, R. Morris, et al. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[39] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A Data-Oriented (and Beyond) Network Architecture. In *Proceedings of the ACM SIGCOMM Conference*, Kyoto, Japan, August 2007.

[40] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turn. OpenFlow: Enabling Innovation in Campus Networks. *ACM Computer Communications Review*, 38(2), April 2008.

[41] J. C. Mogul, P. Yalagandula, J. Tourrilhes, R. McGeer, S. Banerjee, T. Connors, and P. Sharma. API Design Challenges for Open Router Platforms on Proprietary Hardware. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, Calgary, Alberta, Canada, October 2008.

[42] K. Papagiannaki, S. Moon, C. Fraleigh, P. Thiran, and C. Diot. Analysis of Measured Single-Hop Delay from an Operational Backbone Network. In *Proceedings of the IEEE INFOCOM Conference*, New York, NY, USA, June 2002.

[43] C. Partridge, P. P. Carvey, E. Burgess, I. Castineyra, T. Clarke, L. Graham, M. Hathaway, P. Herman, A. King, S. Kohalmi, T. Ma, J. Mcallen, T. Mendez, W. C. Milliken, R. Pettyjohn, J. Rokosz, J. Seeger, M. Sollins, S. Starch, B. Tober, G. D. Troxel, D. Waitzman, and S. Winterble. A 50 Gigabit Per Second IP Router. *IEEE/ACM Transactions on Networking*, 6(3), June 1998.

[44] PIC-SIG. PCI Express Base 2.0 Specification, 2007. http://www.pcisig.com/specifications/pciexpress/base2.

[45] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, October 2001.

[46] J. Turner, P. Crowley, J. Dehart, A. Freestone, B. Heller, F. Kuhms, S. Kumar, J. Lockwood, J. Lu, M. Wilson, C. Wiseman, and D. Zar. Supercharging PlanetLab – A High Performance, Multi-Application, Overlay Network Platform. In *Proceedings of the ACM SIGCOMM Conference*, Kyoto, Japan, August 2007.

[47] L. Valiant and G. Brebner. Universal Schemes for Parallel Communication. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, Milwaukee, WI, USA, June 1981.

[48] B. Veal and A. Foong. Performance Scalability of a Multi-Core Web Server. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Orlando, FL, USA, December 2007.

[49] R. Zhang-Shen and N. McKeown. On Direct Routing in the Valiant Load-Balancing Architecture. In *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM)*, St Louis, MO, USA, November 2005.

[50] L. Zhao, Y. Luo, L. Bhuyan, and R. Iyer. SpliceNP: A TCP Splicer using a Network Processor. In *Proceedings of the ACM Symposium on Architectures for Networking and Communications Systems (ANCS)*, Princeton, NJ, USA, October 2005.