DoubleClick:
Boosting the Performance of Click Modular Router

김 준기 (金駿起 Kim, Joongi)
전산학과
Department of Computer Science

KAIST
2012
DoubleClick:
Click 라우터의 성능 향상에 관한 연구

DoubleClick:
Boosting the Performance of Click Modular Router
DoubleClick:
Boosting the Performance of Click Modular Router

Major Advisor : Professor Moon, Sue Bok
Co-Advisor     : Professor Park, KyoungSoo
by
Kim, Joongi
Department of Computer Science
KAIST

A thesis submitted to the faculty of KAIST in partial fulfillment of the requirements for the degree of in the Department of Computer Science. The study was conducted in accordance with Code of Research Ethics

2011. 11. 25.

Approved by
Professor Moon, Sue Bok
[Major Advisor]

Declaration of Ethical Conduct in Research: I, as a graduate student of KAIST, hereby declare that I have not committed any acts that may damage the credibility of my research. These include, but are not limited to: falsification, thesis written by someone else, distortion of research findings or plagiarism. I affirm that my thesis contains honest conclusions based on my own careful research under the guidance of my thesis advisor.
DoubleClick:
Click 라우터의 성능 향상에 관한 연구

김 준기

위 논문은 한국과학기술원 석사학위논문으로 학위논문심사위원회에서 심사 통과하였음.

2011년 12월 20일

심사위원장 문 수 복 (인)
심사위원 박 경 수 (인)
심사위원 신 인 식 (인)
ABSTRACT

The pervasive deployment of the Internet has increased demands for high-performance and low management costs for network equipments such as routers. Software routers have been promising programmability on top of general-purpose commodity hardware, while dedicated hardware routers have offered extreme performance sacrificing extensibility with high costs. Click modular router has introduced a fine-grained modular architecture, but it and other successors lack the performance despite of recent hardware developments. The reason for low performance of Click are I/O overheads from the Linux networking stack, no concerns for multi-core CPUs and multi-queue NICs, and overheads of fine-grained modular architecture. Based on our insight into PacketShader, which is the performance record holder among existing software routers, we employ the same design principles to Click to overcome its limitations. First, we apply batching at packet I/O using the packet I/O engine from PacketShader to eliminate overheads from the Linux networking stack. Second, we modify the threading model of Click for maximum utilization of multi-core CPUs. Third, we apply batching at packet processing to reduce modularity overheads. We demonstrate our modified Click named DoubleClick and show that its performance reaches close to that of PacketShader while keeping Click’s modularity.
Contents

Abstract ......................................................... i
Contents ......................................................... ii
List of Figures .................................................. iii

Chapter 1. Introduction ......................................... 1

Chapter 2. Background ........................................... 3
  2.1 Batching as a Primary System Design Principle .......... 3
  2.2 NUMA Architecture ........................................ 4
  2.3 The Packet I/O Engine ...................................... 5
  2.4 Click Modular Router ...................................... 6

Chapter 3. DoubleClick ........................................... 8
  3.1 Performance Limitations of Click .......................... 8
  3.2 Our Approach .............................................. 9
  3.3 Extending Click Modular Router ......................... 9
    3.3.1 Multiquene-aware Batched Packet I/O .............. 9
    3.3.2 Multi-processor Scalability ......................... 10
    3.3.3 Batched Packet Processing ......................... 11
  3.4 Implementation .......................................... 11
    3.4.1 IPv4 Router .......................................... 11

Chapter 4. Evaluation ........................................... 13
  4.1 System Setup ............................................ 13
  4.2 Putting All Extensions Together ......................... 13
  4.3 Effect of Batching at Packet Processing ................ 14
  4.4 Latency ................................................ 15

Chapter 5. Future Work .......................................... 16

Chapter 6. Conclusion .......................................... 17

References ................................................... 18

Summary (in Korean) ........................................... 19
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>How batching improves the performance</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>Comparison of a traditional shared bus system and a NUMA system</td>
<td>4</td>
</tr>
<tr>
<td>2.3</td>
<td>The software architecture of the packet I/O engine</td>
<td>5</td>
</tr>
<tr>
<td>2.4</td>
<td>An example of Click processing paths giving the idea of Click’s threading model [9]</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>Comparison of Linux driver and the IO engine in handling multi-queue NICs. It shows how Click can use more CPUs by accessing queues instead of devices.</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>A partial view of the IP router configuration used in DoubleClick. Taken from [9].</td>
<td>12</td>
</tr>
<tr>
<td>4.1</td>
<td>The hardware diagram for our DoubleClick machine</td>
<td>13</td>
</tr>
<tr>
<td>4.2</td>
<td>The throughput comparison by the steps of extending Click. All uses the same IP router configuration.</td>
<td>14</td>
</tr>
<tr>
<td>4.3</td>
<td>The isolated effects of varied batch sizes in pack processing. The absolute values are meaningless.</td>
<td>14</td>
</tr>
</tbody>
</table>
Chapter 1. Introduction

Today the Internet is connecting every electronic device tighter and wider than ever, including smartphones, TVs, and portable multimedia players as well as traditional desktop PCs and dedicated servers. Its ubiquitous deployment and increasing complexity requires more bandwidth and management efforts. From multinational enterprises to local companies, the demands for high-performance, flexible and secure network equipments such as routers, firewalls, and switches are increasing at a fast pace. Newly built datacenters want virtualized and prioritized networks to guarantee the performance SLAs (service level agreements) of their web and cloud services. This trend makes it harder to develop new functions and protocols and to make them scale along the increasing line rates [12].

Software routers have been promising programmability on top of general-purpose commodity hardware, while dedicated hardware routers have provided extreme performance sacrificing cheap price and extensibility. The Click modular router [3,9] is a seminal work in this context, which has suggested a fine-grained modular architecture for packet processing. The shortcoming of software routers is lack of performance (at most 1-3 Gbps) despite of recent breakthrough in hardware technology such as commoditized multi-core systems and high-speed NICs (network interface cards). There have been several attempts to push the performance limit beyond 10 Gbps, which had been considered as the unattainable performance in software routers. RouteBricks [4] showed the potential of horizontal scaling of the Click modular router by reaching 35 Gbps with a clustered setup of four identical machines. Nevertheless, it achieves only 8.9 Gbps on a single machine, which is a even less capability than a single 10G NIC port. A previous work PacketShader [7] is the first software router prototype that reached 40 Gbps on a single machine. Its major contributions are (i) the packet I/O engine for batching and (ii) acceleration of common packet processing operations with GPU. Although PacketShader is holding the highest performance record currently, its implementation is monolithic due to aggressive hardware-specific optimizations. We need to manually stitch all the related codes into a single application in PacketShader. This limits programmability and reusability of the protocol codes to build more complex routers. In constrast, Click’s fine-grained modules offer a maximum flexibility but limit the performance. The flexibility of Click has enabled the researchers to actively add supports for new protocols which was not available at the time of the first release of Click, such as WiFi.

In the mean time, hybrid approaches that use programmable hardware platforms such as FPGA (field programmable gate array) have been proposed. SwitchBlade [2] is a modular framework for custom networking protocols built on top of NetFPGA. However, NetFPGA is still considered expensive ($1,600 for a single card with an integrated Ethernet switching chip and 4x 10G ports [1,10]). Another problem of FPGA-based approaches is that programming on it with hardware description language (e.g., Verilog) has a steep learning
curve for average software engineers. ServerSwitch [11] uses a custom hardware NIC including an Ethernet switching chip for faster data-plane performance. The price of a ServerSwitch card ($400 in the paper) is comparable to that of current Intel 10G NICs ($500), but the speed is limited to a few Gbps since they used only 1Gbps ports.

From the above related work, we think the future flexible router solutions should take the form of software routers. In the perspective of software routers, we observe that batching is the key implementation technique in cost-effective high-performance systems. To keep the advantage of software routers such as flexibility, we start from the modular design of Click in our work. To improve the performance, we employ batching as the primary design principle from the insights into PacketShader.

The question is how much performance we can achieve by combining the major advantages of both Click and PacketShader: modularity and batching. We identify the reasons for low performance of Click as: (i) overheads of the Linux networking stack, (ii) limited concerns for multi-processor/multi-core architectures and multi-queue supports in high-speed network cards, and (iii) overheads of its fine-grained modular architecture. We improve the performance of Click by applying batching at packet I/O and packet processing. We make it scalable with multi-core systems by modifying Click’s threading model.

In this work, we present a performance-improved version of Click modular router, “DoubleClick”. The name means we have doubled and doubled again its original performance, from about 7 Gbps with the first prototype using batched packet I/O, to 28 Gbps with modified threading model and batched packet processing added. The main contribution of this work is that we confirm batching is the general key principle to implement a high-performance software router which supports more than 10 Gbps lines. We extend Click’s elements to handle batch of packets efficiently, and give a perspective on rooms to overcome limitations of the original Click architecture for modern multi-processor systems. From this work, we finally suggest design guidelines for the future software routers.

This paper is organized as follows. We explore the background technologies at Chapter 2. The design and implementation is explained at Chapter 3 and the evaluation results at Chapter 4. We discuss other framework issues such as adding GPUs to DoubleClick and heterogeneous parallelization models in Chapter 5. Finally, we summarize the contribution of this work at Chapter 6.
Chapter 2. Background

2.1 Batching as a Primary System Design Principle

Batching is a general optimization technique in computer systems that aggregates repeated invocation of a task by serializing the input data. It does not reduce the actual computation time required for each repetition of the task itself, but achieves better performance by reducing the invocation and interfacing costs. Thus batching is effective when applied to the tasks that have relatively high invocation costs than execution costs.

In most computer systems the overheads of repeated tasks mainly come from function calls. At low level, a function call involves manipulation of the stack to pass argument variables. If it involves a system call, the CPU must perform the costly kernel/user mode switching for privilege separation. If the function have to copy large amount of the input data to another location (e.g., from the kernel space to the user space), the invocation cost increases even higher. In networked systems, the function call overheads may include the round-trip time to a remote callee.

Caching is one of the most common solutions to the overheads of repeated execution in both hardware and software. It effectively reduces the repeated invocation costs of the same function with the localized input datasets by memoization of previous results. The caveat is that it cannot handle continuously changing or streamed data because the cache is invalidated at the every moment.

Here batching kicks in. A software designer can predict the temporal patterns of input and locality characteristics of data depending on the application she builds. From the prediction, we can establish a “batching strategy”. We can use windowing techniques to process streamed data like network packets by slicing the
stream into multiple chunks and treating each chunk as a batch. For throughput-oriented applications, we use larger batch sizes in sacrifice of the latency. If we have heterogeneous processors in the system, then we can choose the processor to process the given data depending on the input batch size at the moment. For example, traditional fat-core CPUs are adequate for small batches in favor of lower latency while massively-parallel processors for large batches in favor of higher throughput.

PacketShader [7] has showed that batching is essential at network packet processing.

### 2.2 NUMA Architecture

As the scale of computer systems increases in terms of memory size and computation power, multi-core processors and multi-processor systems have appeared as response to facing the limit of clock speed races. The trend of horizontal scaling involves elimination of single points of bottlenecks such as the shared system bus and cache bouncing.

The single processor systems use a system-wide shared bus (as known as front-side bus) between the CPU and the main memory as shown in Figure 3.1a. There is an intermediate chip called NorthBridge which interconnects the main memory, high-speed I/O devices such as GPUs, and the CPU. Other peripherals were connected via SouthBridge which has an internal connection to NorthBridge. As multi-core processors become popularized, the front-side bus and NorthBridge become the bottleneck.

The multi-processor systems use the non-uniform memory access (NUMA) architecture as in Figure 3.1b. Each CPU has its own memory controller which corresponds to shared NorthBridge in the single processor architecture. The main memory is segmented to each CPU so that they provide dedicated access to each CPU. Other I/O devices are connected via I/O hubs (IOH) which are also dedicated to their owner CPUs. There is a point-to-point interconnection among multiple IOHs and CPUs to provide each CPU with a transparent access to devices and memory which belongs to other CPUs. Software developers does not have to add anything to execute existing codes due to its transparency. However, to scale linearly as the number of processor
increases, the workloads must be localized to the CPU-memory pairs because the access latency to memory on remote sockets is higher (about 2×).

2.3 The Packet I/O Engine

The packet I/O engine from PacketShader is a high-performance device driver for network interface cards based on Intel 82598/82599 chipset [8]. It is a rewritten version of Intel IXGBE driver modified for batched packet I/O. It provides a set of user-level APIs for throughput-oriented network applications. The two main contributions of the packet I/O engine is batching and multi-queue aware device handling.

The packet receiver/sender APIs expose the direct access to multiple queues for each NIC via the “pack” data structure containing multiple packets in a continuous buffer. Continuous buffers enable aggressive prefetching and reduced cache misses in sequential processing of each packet in a pack for most cases. All individual packets are passed and processed as Ethernet frames.

The device handling APIs expose the concept of multiple RX/TX queues instead of devices as a whole. Modern high-speed network interface cards provide multiple queues in the kernel which are written and read by the device directly and have unique IRQ IDs. This enables isolation of packet buffers for each CPU core in multi-core and multi-processor systems. Hence it can eliminate lock contention and cache bouncing by associating each queue with a different processor. NICs evenly spread over the received packets using the RSS (Receive-Side Scaling) technique which uses the hash value of packet headers to select a queue for a packet. By this mechanism, all packets in a single flow goes into the same queue, and thus reordering of packets within a flow does not occur.

Figure 2.3: The software architecture of the packet I/O engine.
2.4 Click Modular Router

The Click modular router provides a modular software architecture for generic packet processing with
arbitrary compositions written in its own configuration language. It comes with a number of elements that
receives, transmits, stores and modifies a packet. The basic unit (module) of processing is called elements.
Each element has input and output ports which accepts and emits a single packet at a time. An element
usually defines an indivisible operation such as decrementing TTL of IP packets, but we can put arbitrarily
complex processing into it.

There are several types of elements: packet sources, packet sinks, packets modifiers, packet checkers,
runers, queues, and etc. The typical example of packet sources and sinks are FromDevice andToDevice
which reads and writes packets from/to an Ethernet device. Another type of packet sink is Drop which
discards the input packet and frees its buffer. The processing starts from a packet source and ends at a
packet sink.

From Figure 2.4, we can grasp the idea of Click’s approach to multi-threading, which associates a thread
with a processing path. Processing paths are classified into two categories: push and pull. Push paths are
initiated by a packet source element and the packet is passed downstream to the destination element. Pull
paths are initiated by a packet sink element and each element in the path asks whether a packet is available
and retrieve it. For most elements all ports follow the same direction, so we call such elements as push
elements or pull elements. Pushing is designed for repetetive tasks such as polling a NIC and processing
packets while pulling is designed for tasks triggered by events because they are initiated when the destination
element is ready to process further inputs. For example, ToDevice element that transmit a packet via a
NIC is scheduled to execute when the router detects the NIC is ready to send. A few exceptions are several
Queue and PullToPush elements which connects a push processing path and a pull processing path. A
typical Queue element stores a packet passed from the push processing path, and returns them as requested
by the pull processing path afterwards. PullToPush does the reverse. In traditional Click, we can use either
the adaptive scheduler and the static scheduler. The static scheduler manually associates a specific thread to

---

1In PacketShader, we had used the term “chunk” for this data structure. We changed it to “pack” to avoid confusion with the
concept of ‘slices from larger data unit’ since we intend to represent ‘an aggregated batch of smaller units’ (packet).
a specific processing path, and it has been recommended to allocate different threads for the connected push and pull processing paths since a packet arrived via a port is usually transmitted via another different port so it is better to avoid crossed access to NICs from the same CPU. The more details on multithreading of Click are explained in [3].
Chapter 3. DoubleClick

In this section we analyze the performance limitation of Click and extend Click step by step to solve them.

3.1 Performance Limitations of Click

1. Low I/O performance. The user-level Click uses the Linux networking stack. It imposes the overheads of per-packet memory allocation. According to [7], the skb and slab memory subsystem in the Linux kernel consume more than 60% of CPU cycles when a bare system receives 10 Gbps of 64B packets.

2. Limited multi-core scalability. The multithreading model of Click has several performance limitations.

   • Resource-intensive multithreading model. The connected push and pull processing paths run in different threads and they are more likely to run in different processors. This means that a packet must travel multiple processors in the system before being transmitted. In multi-core processors, cache invalidation brings high latency to memory access to synchronize cache contents local to other cores. The relative costs of such cache bouncing is higher compared to traditional parallelizable computations because our workload is about processing many small packets with low latency repeatedly. Queues accessed by different threads also require extra synchronization.

   • No supports for multi-queue NICs. Recent high-speed NICs have a configurable number of isolated RX and TX queues as kernel buffers. The driver spreads received packets over those queues by the hash values of packet headers. This enables contention-free packet processing in multi-core systems and linear scaling with the number of cores. However, the packet I/O elements (FromDevice and ToDevice) interact with devices, not the queues of devices. This means Click can only utilize the same number of CPU cores to the number of NICs.

   • No concerns for NUMA architecture. Multi-processor systems and the NUMA architecture for them are now commoditized. Click’s threading mechanism has no concerns for such environments at all, and thus has extra overheads on them due to mixed memory access latencies.

3. Overheads of modularity. Click elements define simple individual packet operations. This improves flexibility and reusability of the codes, but increases the overheads of function calls. 10 Gbps traffic means the handlers of all elements in the processing path are invoked more than 14 million times per second. In our preliminary evaluation, it consumes more than 20% of CPU cycles.
3.2 Our Approach

Here are our approaches to increase the performance of Click as follows:

1. **Apply batching at the packet I/O.** We attach the packet I/O engine from PacketShader to eliminate I/O overheads by batching frequent I/O operations and bypassing the Linux networking stack.

2. **Make Click to access NIC queues directly.** Another benefit to use packet I/O engine is direct access to NIC queues. We modify the packet I/O elements to interact with queues using the user-level API of the packet I/O engine. In this way, we can fully utilize arbitrary number of CPU core by configuring the number of queues per NIC.

3. **Use SMP threading model.** Click supports multi-threading but it works in granularity of push/pull processing paths. We unify those two types of processing paths into push-only and dedicate a thread to run a whole push processing path. Each thread (equivalent to a processing path) consumes packets from the NIC queue associated with it.

4. **Affinitize threads considering NUMA architecture.** The resources such as memory and devices accessed by a thread and the thread should reside in the same node to avoid extra memory access latency overheads. The interrupts should be delivered to the thread of which is responsible to eliminate context switching overheads. To avoid scheduling a thread on different NUMA nodes as time goes and get a steady performance, we need to fix the mapping of threads to cores. We modify Click to pin a core to run a thread.

5. **Apply batching at the packet processing.** We believe that batching is essential to achieve high performance and it should be applied to everywhere possible. The remaining part of Click to apply batching is its elements. We extend existing elements to handle sequential batches of packets at once instead of individual packets.

3.3 Extending Click Modular Router

3.3.1 Multiqueue-aware Batched Packet I/O

We have attached the packet I/O engine for batched packet I/O and multi-queue awareness to overcome slow I/O performance in the user-level Click (the approach 1 and 2). We think the future software routers should avoid kernel-level packet processing because it may cause starvation of user-level processes in overloaded conditions due to higher scheduling priority of the kernel and irreversible system corruption due to unexpected bugs. In the user-level packet processing, however, we can seamlessly integrate 3rd party libraries and develop experimental features with ease due to rich and friendly debugging environments.
Figure 3.1: Comparison of Linux driver and the IO engine in handling multi-queue NICs. It shows how Click can use more CPUs by accessing queues instead of devices.

To do this, FromDevice and ToDevice is converted to use the user-level API of the packet I/O engine.\(^1\) Each FromDevice element accesses one of the RX queues associated with the device instead of libpcap or raw sockets which can interact with a device as a whole as in Figure 3.1. This enables use of more number of CPU cores than the number of NICs by changing the number of RX queues per NIC, and hence we can utilize the full computation power in multi-core systems with multi-queue NICs.

To support batched packet I/O, we have added two operation modes to the new FromDevice and ToDevice elements: batching mode and compatible mode.

- **Batching mode**: This mode is used at pack processing configurations. FromDevice receives a pack of packets and emit as it is. All elements in the processing path should be able to handle the pack instead of packet instances. ToDevice transmits the pack it receives as it is. There is no copying in the whole processing path if we do not include elements that perform cloning.

- **Compatible mode**: This mode is for unmodified Click elements to work with the packet I/O engine. It creates multiple packet instances with the pointers to the actual continuous buffer of the received pack. The packet instances share the buffer of the received pack until it arrives at ToDevice elements. ToDevice copies the content of packets to a new pack and aggregates packet until the number of received packets reach the configured TX batch size.

### 3.3.2 Multi-processor Scalability

We have modified RouterThread implementation to set thread affinity at creation of each router thread (the approach 3 and 4). The thread creation routine also calls NUMA-binding function to make all memory allocated after the call to reside at the same node. We have inserted StaticThreadSched for each FromDevice elements in the Click configurations to utilize this extension. In the configuration, users

---

\(^1\)There is another type of packet RX element called PollDevice which polls the device instead of waiting interrupts. The converted FromDevice implements the same method used in PacketShader, dynamically changing polling and blocking I/O.
can indicate which CPU core execute which part of the processing path, CPU 0 runs a \texttt{FromDevice} element associated with the queue 0 of the NIC 0 and its descendant elements for example. The configuration for each thread is identical so that all threads executes the same router pipeline with different packs from the associated NIC queues.

One limitation of our modification is lack of NUMA-aware memory binding because Click’s element initialization procedure does not allow knowing the thread associated with a processing path which will execute it and memory allocation of elements happens before the threads are created. We leave this as a future improvement.

### 3.3.3 Batched Packet Processing

At last, we apply batch processing of packets to Click elements (the approach 5). To run the I/O elements (\texttt{FromDevice} and \texttt{ToDevice}) in the batching mode, all intermediate elements should be extended to handle packs as their input and output, instead of individual packets. We call this mode as \textit{pack processing}.

For most elements which have only one input and one output ports, we simply add an iteration loop at the processing handlers.

However, the non-trivial case is the elements which have multiple output ports, like \texttt{LookupIPRoute}. The output connected to a \texttt{Drop} element can be implemented as an operation that marks a packet in the buffer of the input pack as ‘deleted’ instead of freeing the memory because the intermediate elements within a processing path share the same continuous buffer containing all packets and thus we cannot free individual packets. The outputs are connected to other processing elements requires split of the input pack since individual packet in a pack may have different destinations depending on the processing result. In such cases, the element prepares $N$ buckets (dummy packs) to accumulate packets going to $N$ output ports. As the iteration over the input pack goes on, each bucket stores the packets like a queue. All buckets share the same buffer with the input pack, and only the pointer to the actual packets are copied. If processing of all packets in the input pack finishes, the element turns over the buckets as input packs to destination elements.

### 3.4 Implementation

#### 3.4.1 IPv4 Router

We implemented and wrote the configuration for a simplified IPv4 router as in Fig 3.2. It omits ARP querying and routing table update mechanisms as known as the control-plane. The IP lookup algorithm is DIR-24-8-BASIC [6]. We let all CPU cores (8 cores in our hardware configuration) execute the same task set: packet RX and TX, IP header check, IP route lookup, and updating the TTL and source/destination ethernet addresses. For RX and TX, we use the converted \texttt{FromDevice} and \texttt{ToDevice} elements. Each threads has its own TX queue for each NIC and shares a RX queue with other cores in the same NUM
node. For example, core 2 in the node 1 is attached to the RX queue 2 of the NIC 2 and 3 which reside in the same node. It outputs the packets via the TX queue 2 of each four NICs. The node crossing at TX is inevitable because packets are forwarded to arbitrary NIC ports.

The execution flow is same to PacketShader which sends out the packets after processing them all without queuing. The basic loop of each thread is (i) a FromDevice element receives a pack of packet, (ii) converts it to multiple packets or split into smaller packs, (iii) initiates the push processing (the processing path includes many elements such as IP header checks and IP route lookup), (iv) the ToDevice element at the last gathers packets into its own pack or transmit the pack it received directly.
Chapter 4. Evaluation

All evaluations are the average of total 12 data points generated from 3 repeated executions and 4 records per repetition. The values are recorded after 20 seconds of warming up to reduce artifacts by other programs running. The metrics are total and per-NIC TX throughputs, CPU loads, and the round-trip latency. CPU loads are omitted because all experiment conditions appears to use all CPU cores at full load.

4.1 System Setup

The system uses two Intel Xeon X5550 2.66 GHz processors (quad-core) with 1,333 MHz 12GB DDR3 memory. The two NUMA nodes have one CPU each. It has two dual-port Intel 82599 X520-DA2 10GbE NICs in each node as shown in Figure 4.1. The maximum aggregate capacity of the system is 40 Gbps.

4.2 Putting All Extensions Together

Figure 4.2 shows how much the performance is improved as our extensions are applied step by step. Batching at the packet I/O gives the baseline performance about 7 Gbps. NUMA-aware thread affinity doubles the performance for packet sizes larger than 1, reaching 16 Gbps. Batching at the packet processing gives sub-linear performance scaling with the pack size. The peak throughput reaches about 28 Gbps at the pack size 512.

Here are the number of previous work in software router to compare: (i) RouteBricks has reached 8.7 Gbps with a single machine. (ii) PacketShader has achieved 40 Gbps with a single machine. Both are the performance of IPv4 routing. We observe that Click has more potential than expected in RouteBricks. We need to think of the last mile to defeat PacketShader. It might be the part of GPUs.
**4.3 Effect of Batching at Packet Processing**

Figure 4.3 shows the isolated effects of batching at packet processing. We did this experiment to know how important batch processing at elements is. The pack size for elements 1 means intermediate elements processes packet-by-packet via the pack data structure. The pack size for elements 32 means they process 32 packets in the given pack at once before passing them to next elements. We have replaced ToDevice to an emulated version which has negligible processing cost because it just does nothing. The RX pack size is fixed to 1024 packets per pack regardless of the pack size for elements.

To do that, we have added one more step to FromDevice that splits a received pack into multiple smaller packs, for example, 1024 packets of a pack is split into 32 packs containing 32 packets for each. Then the variation in this graph shows the only element batching effect. This result shows batch processing is a general technique.
4.4 Latency

In networking, the latency is critical for interactive applications. We have measured the processing latency of DoubleClick by sending and capturing magic packets in the packet generator in the condition that there is no packet misses. Still this measurement method may have more variations compared to hardware-based latency measurements, so we take the largest latency value possible in each pack received at the packet generator.

On average, the IO engine only version showed 230 us. The final pack processing version showed 180 us. This is counter-intuitive because increased use of batching and increased batch sizes typically increase the latency. We think that high-speed networks running at 10/100 Gbps quickly fill the packs and the computation room gained by batching beats the intrinsic latency overhead.
Chapter 5. Future Work

Our ultimate goal is to build a high-performance modular architecture for software routers. Still, we have a lot of work to achieve it. The next sub-goals include:

- **Solving the path-diversity problem:** We call the size cut-down of packs after passing the elements with $N$ outputs the path-diversity problem. In more complex router configurations, repeated size cut-down will reduce the performance gain from batching. To prevent it, we need to aggregate small packs into larger packs somewhere in a way memory copy overheads is minimized.

- **Evaluation of different parallelization models:** SMP vs. dynamic task scheduling. Both PacketShader and DoubleClick uses the SMP model, meaning that all CPU cores perform identical tasks: packet I/O and computation. What if each CPU core runs different tasks? It is known that passing packets around multiple cores are not desirable due to increased cache misses [5]. However, it may be a good deal for offloading imbalanced traffic if we replace the tasks assigned to other processors with the overloaded task. In this sense, we could add a scheduling algorithm that change the assignmet of tasks to CPU cores dynamically depending on the traffic pattern.

- **Addition of a computation abstraction layer for heterogeneous processors:** We want to incorporate GPUs as the computation accelerators as PacketShader did because the current bottleneck is CPU. The major technical challenges are asynchronous computation and data copy overheads between disjoint memory address spaces. Along that, multiple processors sharing GPUs concurrently will cause synchronization overheads and thus decreases the utilization of the GPU’s full computer power. Its design must implement low latency and in-order processing schemes, and address load balancing problems.

- **Searching possibilities for even higher performance:** (e.g., 100 Gbps on a single machine) We expect using the latest hardware technology such as heterogeneous multicore processors would open new challenges and opportunities for further performance improvements.

- **Integration of the control-plane:** Most evaluations on software routers have focused on the data-plane performance without the control-plane. Since the control-plane changes the internal states of a router and may cause non-trivial performance impacts, the data-plane should be evaluated with it.

We are currently working on the computation abstraction layer and testing the dynamic task scheduling model. Another plan is to support GPUs again in future versions with a new abstraction layer that handles all subtle issues related to parallelized modular framework.
Chapter 6. Conclusion

The long-cherished goal of software routers is to achieve both flexibility and high performance. From the existing software routers, we could observe that batching is the key for high performance and modularity is desirable since it can spread a framework widely. We have presented DoubleClick by extending the Click modular router to exploit batching and hardware-specific optimizations such as the NUMA architecture and multiqueue NICs. The result shows that the performance when all techniques are applied is increased at four folds (7 Gbps to 28 Gbps) compared to the case when we naively applied I/O batching only. Through the series of extension and evaluation, we confirm that batching is essential for software routers to go beyond 10 Gbps with fine-grained modular architectures. From this work, we insist that the design guidelines for future software routers must include batching everywhere possible as the primary design principle as well as intrinsic supports for the latest hardware technologies.
References


Summary

DoubleClick: Boosting the Performance of Click Modular Router

인터넷이 광범위하게 사용됨에 따라 새로운 기능을 손쉽게 추가할 수 있는 유연함을 갖춘 고성능 네트워크 장비에 대한 요구가 증가하고 있다. 소프트웨어 라우터는 범용 하드웨어를 사용해 저렴한 비용으로 최대한의 프로그래밍의 장점을 제공하는 것이 그 목표이다. 이는 전용 하드웨어 라우터가 가격과 확장성

을 희생하고 최대한의 성능을 제공하는 방향으로 발전해온 것과 대비된다. Click 라우터는 소프트웨어

라우터 중 모듈화 구조를 처음 소개하여 그 런어난 확장성 덕분에 많은 연구자들이 사용했으나, 낮은 성능이 그 단점으로 지적되어왔다. Click의 낮은 성능은 Linux networking stack이 가지는 오버헤드, 멀티코어 CPU와 다중 큐 네트워크에 대한 지원 미비, 그리고 모듈화 구조에 따른 오버헤드로부터

기인한다. 이러한 문제점을 해결하기 위해, 처음으로 40 Gbps의 벽을 뚫은 PacketShader 연구로부터

일괄처리(batch processing)가 성능 향상에 가장 핵심적인 요소임을 파악하고 이를 Click에 다양한 방

법으로 적용하였다. 첫번째는 패킷 I/O에 일괄처리 기법을 적용하고, 두번째는 threading model을 변

경하여 멀티코어 환경과 다중 큐 네트워크에 대한 지원을 강화하고, 세번째로는 패킷 처리 과정에

일괄처리 기법을 적용하였다. 본 연구에서는 이렇게 Click의 성능 개선 버전인 “DoubleClick” 라우터

를 개발하였고, 그 성능을 평가하여 소프트웨어 라우터가 Click의 유연한 모듈화 구조를 유지하면서도

PacketShaper에 필적하는 성능을 낼 수 있음을 보였다.
감 사의 글

이 논문을 작성하기까지 물심양면 항상 신경써주시고 이 연구가 나아가야 할 방향을 보다 넓게 바라볼 수 있도록 지도해주신 문수복 교수님께 먼저 깊은 감사의 뜻을 표합니다. 또한 뛰어난 시스템 연구자로서 문수복 교수님과 함께 실험 설계와 결과 해석에 많은 기여를 해주신 박경수 교수님께도 감사의 말씀을 전합니다. 이 과정에서 교수님들 못지 않게 글쓰기와 프로그래밍 작업을 모두 아우르는 세부 사항에 대해 조언과 격려를 아끼지 않으신 연구실 선배 장건 형과 한상진 형에게도 감사드립니다.

시간이 부족한 가운데에도 논문에 필요한 실험 수행과 프로그래밍 작업에 많은 도움을 주고 실험환경 세팅 등 귀찮은 일들을 도맡아주신 연구실 동료 성구 형의 도움도 물론 배드릴 수 없을 것입니다. 한편 부득이한 스트레스 상황에서도 항상 즐겁게 연구실 생활을 할 수 있도록 함께 밤도 먹고 게임도 같이 한 연구실 선후배 분들께도 감사합니다. 비록 다른 분야를 공부하지만 논문 준비 과정에서 색다른 관점의 코멘트를 남겨준 동지한 현우 형과 호성 형, 나카로운 장현 형, 또 항상 질문이 많은 윤성이와 제이치는 정인이까지 모두 큰 도움이 되었습니다. 마지막으로, 제가 혼들어할 때 항상 따뜻한 격려의 말과 편지로 계속 앞으로 나아갈 수 있도록 이끌어주신 어머니, 그리고 항상 든든하게 가정을 지키며 제가 아무런 걱정 없이 연구에 전념할 수 있도록 해주신 아버지와 인기 형에게도 이 자리로 빛어고맙다는 말씀을 전하고 싶습니다.
이력서

이름: 김준기
생년월일: 1987년 5월 12일
출생지: 대한민국 경기도 과천시
본적지: 대한민국 서울특별시 영등포구
주소: 대한민국 경기도 용인시
E-mail 주소: joongi@an.kaist.ac.kr

학력
2003. 3. – 2005. 2. 경기과학고등학교
2005. 3. – 2010. 2. 한국과학기술원 전산학과 (B.S.)
2010. 3. – 2012. 2. 한국과학기술원 전산학과 (M.S.)

경력
2006. 4. – 현재 TNF/Needlworks (Tatter Network Foundation) 개발자
2009. 6. – 2010. 1. NexR 인턴 및 프리랜서 SW Engineer
2010. 2. – 2010. 5. KAIST CS101 Introduction to Programming 조교
2010. 2. – 2010. 5. KAIST CS220 Programming Principles 조교
2010. 10. – 현재 Popong (Open Politics Engineering) 데이터플랫폼 개발자

연구 업적
2. **Joongi Kim**, Keon Jang, Sangjin Han, Kyongsoo Park, Sue Moon, *Dynamic Forwarding Table Management for High-speed GPU-based Software Routers* [poster], 9th USENIX Symposium on Operating Systems Design and Implementation (2010), Vancouver, Canada
