The Power of Batching in the Click Modular Router

Joongi Kim[†] Seonggu Huh[†] Keon Jang[*] KyoungSoo Park[‡] Sue Moon[†]

[†]Department of Computer Science, KAIST, Korea
{joongi, seonggu}@an.kaist.ac.kr, sbmoon@kaist.edu

[‡]Department of Electrical Engineering, KAIST, Korea
kyoungsoo@ee.kaist.ac.kr

[*]Microsoft Research, Cambridge, UK
keonjang@microsoft.com

Abstract

The Click modular router has been one of the most popular software router platforms for rapid prototyping and new protocol development. Unfortunately, its internal architecture has not caught up with recent hardware advancements, and the performance remains sub-optimal in high-speed networks despite its benefit of flexible module composition.

In this work, we identify the performance bottlenecks of the existing Click router and extend it to scale with modern computer systems. Our improvements focus on both I/O and computation batching, and include various optimizations for multi-core systems and multi-queue network cards. We find that these techniques improve the performance by almost a factor of 10, and the maximum throughput reaches 28 Gbps of minimum-sized IPv4 packet forwarding speed on a single machine.

1 Introduction

The Click modular router [12] has been one of the most popular software platforms that allow flexible composition of packet-processing functionalities. It has been used in a number of application prototypes such as a router for a future Internet architecture [9], redundant traffic elimination systems [2], a scalable middlebox platform [17], and a cluster-based high-performance software router [8], just to name a few. The key strength of Click is its inherent extensibility of functional components: a new feature can be easily implemented by composing existing and new modules.

APSys '12, July 23-24, 2012, Seoul, S. Korea

While Click's flexible design has satisfied many of the demands for rapid prototyping, its internal architecture has not caught up with recent hardware advancements. In the processor side, multi-core CPUs in a non-uniform memory access (NUMA) architecture have become common. In the I/O side, multi-queue network interface cards (NICs) and high-speed interconnects have become commoditized, which promises high performance packet processing. Yet, the existing Click router does not utilize the modern advancements in hardware technology to its full potential, limiting its effectiveness in the high-speed multi-10 Gbps networks.

There have been various approaches to improve the performance of Click [3, 5, 8, 14, 18], but most of them focus on parallelizing the packet processing workloads on multicore systems. In this work, we explore the benefit of *batching* in the Click software router while preserving the modular architecture. More specifically, we carefully gauge the performance improvements coming from packet I/O batching and computation batching.

Packet I/O batching: the typical performance bottleneck in any high-performance packet processing system on commodity hardware lies in the slow packet reception (RX) speed. The source of the problem is mainly attributed to high per-packet RX overheads, involving the inefficiency of

the networking stack of a general-purpose OS [10]. With I/O batching, the per-packet RX overheads are amortized by reducing the common tasks per each packet and by eliminating the CPU consumption on per-packet memory management.

Computation batching: factoring the common computation (e.g., repetitive function calls) for a group of packets should improve the performance, especially for small packets. The idea is similar to system call batching [19, 20], but we apply it to user-level packet processing. One caveat is that computation batching reduces the per-packet processing flexibility, so the design should strike the balance between the modular architecture and the performance improvement.

For I/O batching, we can choose among several open-source solutions, including PacketShader's packet I/O engine (`psio`) [10], `netmap` [15], and `PF_ring` [1]. They are designed with the same underlying principle that batched access of packet DMA buffers from user applications boosts the I/O performance. Carefully-designed I/O batching not only reduces frequent kernel-user switching, but also eliminates memory allocation overheads (e.g., skbuf). `netmap` and `psio` both focus on efficient user-level application support, where the former provides a hardware-independent interface and the latter is based on an extended Intel NIC driver. However, we find that `netmap` does not perform as much as we expected (explained in section 2.3), so we use `psio` for this work. `PF_ring` targets packet filtering workloads instead of user-level packet I/O, and bears a performance hit due to conformance to existing kernel structures. While it supports direct NIC access for high performance (called DNA), it allows multiple 10 Gbps line speed only when it uses the polling mode, which wastes CPU cycles.

For computation batching, we propose to convert existing Click elements to be friendly to batch processing. The simplest approach would be to convert every element to take and process a batch of packets. We demonstrate the performance improvement using a simple IPv4 router. We find that computation batching gives as much as 75% better performance with 64 B packets compared to the naive integration of `psio` regarding multi-queue support and I/O batching.

2 The Click Modular Router

2.1 Elements

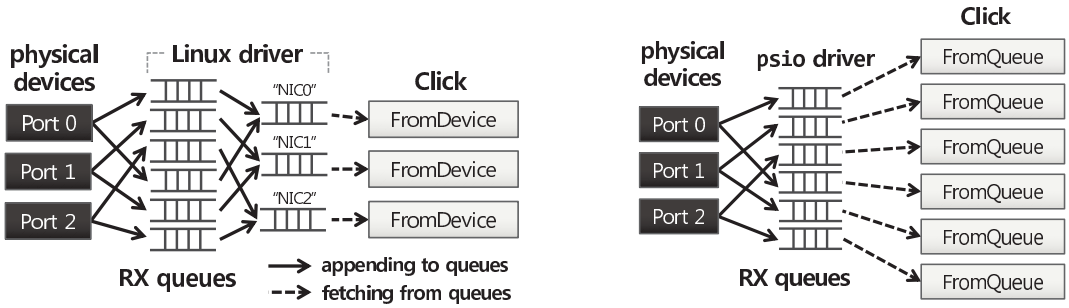
The basic processing abstraction of Click is called *element* [12]. Each element accepts a packet, applies some operation, and directs the packet to another element. An element may have multiple input/output ports that connect to other elements. There are several types of elements: packet sources and sinks, filters, and routers. Packet sources and sinks are the I/O elements interacting with the NICs via OS or a custom interface such as `libpcap`. Packet filters validate the values of several fields in packet contents (*e.g.,* IP checksum and TTL) and drop invalid ones. Packet routers direct each packet to one of their destination elements depending on the computation result. A typical router configuration describes the linkage among these elements with element-specific parameters.

2.2 Threading Model

The execution unit of the multithreaded Click router is called a *processing path* [5]. A processing path is a directed graph of connected elements that are executed in sequence, which is disjoint to other processing paths. There are two types of paths: *push* and *pull* paths. A push processing path starts with a packet source element, which generates a packet and passes it down to other elements in the path via function calls. A pull processing path starts with a packet sink element, which chooses one of its input ports and performs an upcall to retrieve a packet from the path. Between the push and pull paths, there is a queue element that stores packets temporarily. Typical router pipelines consist of a pair of push and pull processing paths. The main loop of each thread repeatedly fetches and executes a path from its private work queue, and the Click router applies either adaptive or static schedulers to manage those queues.

2.3 Performance

There have been many efforts to improve the performance of Click. SMPClick [5] is the first multithreaded implementation of Click, as described in the section 2.2. It distributes workloads to multiple cores by scheduling different elements on different cores. While the general idea is insightful, the communication overhead between cores can under-



(a) Linux driver and per-device I/O elements

(b) `psio` driver and per-queue I/O elements

Figure 1: A comparison of the current and proposed packet I/O schemes for Click.

mine the benefit of pipelining on today's processors [6,7]. I/O batching for Click has brought substantial performance boosts [3, 8]. RouteBricks introduces multi-queue NIC support for the kernel version of Click, showing significant performance improvements [8]. There are other extreme approaches such as using NetFPGA [13, 14] or Network Processors [18] to accelerate the Click router.

The recently-added `netmap` support in Click performs better than the existing `pcap` interface [15]. However, it still does not saturate a single 10 Gbps link. The reported performance using the `libpcap` wrapper atop `netmap` is about 2.8 Gbps with simple packet forwarding with two 10 GbE cards [15]. This number is contrasted by the raw performance of `netmap`, which is 10 Gbps in the same literature. Our measurements using the latest version of `netmap` support show poor performances with less than 1 Gbps with four 10 GbE cards doing packet RX only, and we suspect that the current version of `netmap` does not support multiple NIC queues efficiently. Our work encourages the idea of per-queue I/O elements [8] by showing that the above mismatched performance can be improved. On top of that, we add computation batching for further performance improvement.

3 Design

We apply I/O and computation batching to the Click modular router. We keep the essence of its design, the modular architecture and the configuration language, while making it faster and more scalable.

3.1 Design Goals

We first introduce our high-level design goals:

- *High performance.* The extended Click modular router should be able to sustain a high input rate such as multiple-10 Gbps line rates.
- *Multi-core scalability.* We make Click scalable to the number of available CPU cores. That is, the performance should grow as the number of cores increases.
- *Element API and configuration compatibility.* For easy adoption of our design, users should not have to change their elements and configurations. The changes should be minimal and automated as much as possible.

3.2 Our Approach

Drawing the insights from previous works on high-speed software router designs [6, 10, 11], we take the following approaches.

- *Aggressive batch processing.* Batching amortizes the per-packet processing cost, and it is mandatory to saturate multiple high-speed NICs (*e.g.,* 10 Gbps, which corresponds to approximately 14.8 Mpps for 64B packets). So we apply batching everywhere possible—both for packet I/O and computation.
- *Native multi-queue NIC support.* Modern network cards have the ability to configure multiple DMA buffers (queues) and distribute packets to them by hashing packet headers as known as RSS (receive-side scaling). This enables linear scaling of the packet I/O performance with the number of available cores. We adopt the per-device I/O elements to exploit them.
- *SMP (symmetric multiprocessing) model.* Splitting the router configuration into multiple parts and pipelining them using different cores incur synchronization overheads and compulsory cache misses on packet reads [6]. We clone the router

configuration and make all cores to execute the same configuration with different set of input packets to prevent those problems.

- *NUMA Awareness.* NUMA is a commonly used way to scale up the performance of a single machine, adding multiple physical processors with their local memory. We take care of data and thread placement in NUMA systems to reduce the contention at CPU interconnections and memory controllers [4].

3.3 Key Techniques

I/O Batching. We accomplish packet I/O batching by using `psio`. It passes the received packets in large batches to the user-level Click. Our Click also transmits the packets in large batches. By default, individually-transmitted packets are queued at the end of processing path up to the batch size. When computation batching is applied, the group of packets processed together (called a *pack*) is immediately sent out to NICs. Since the performance heavily depends on the size of a pack (called a batch size) as shown in Section 4, we use a few hundreds packets as a batch size.

Computation Batching. We apply batching to the intermediate elements between the packet source and sink. The received pack is passed down to the processing path, being handled by elements in the middle. Each element takes it and applies their operations by iterating over the packets in the same pack. See Table 1 for details on how we enforce batch processing with various elements with different input/output port numbers. We do not have to take into account push or pull port semantics since we use only push processing paths for our Click configurations.

Multi-queue NIC support. We exploit multi-queue NICs by adopting the per-queue I/O elements. As shown in Figure 1, we add two elements, `FromQueue` and `ToQueue`, which interact with the individual NIC RX/TX queues via `psio`. `FromQueue` operates as a packet source, generating a group of packets by reading the associated RX queue. `ToQueue` works as a packet sink, receiving a pack and writing the packets to the TX queue. The number of packets in a pack can be configured.

NUMA-aware Scheduling. Combining the above techniques step by step, we finally introduce NUMA-aware scheduling. First, we replace Click's `FromDevice` and `ToDevice` elements with `FromQueue` and `ToQueue` elements. Second, we move all intermediate processing elements to the push path from the pull path. This step removes redundant packet queuing at the end of processing path. Third, we set the number of the push processing paths as the number of DMA queues, and set up a static scheduling: one-to-one mapping of RX/TX queues with CPU cores. For NUMA systems, we need to separate RX queues for each node to ensure that the packets reside in the local memory before being moved to output TX queues. This allows at most one NUMA crossing when the packets need to be forward to TX queues in a different node.

# input/outputs	How to apply batching?
1 / 1	Wrap the processing handler with a for-loop to iterate over individual packets in the input pack. (e.g., `Strip`, `CheckIPHeader`)
1 / 2 (one is drop)	Wrap the processing handler with a for-loop and mark dropped packets in the pack. Later elements and `psio`'s TX part will ignore them. (e.g., `DecIPTTL`, `IPGWOptions`)
1 / *n* (router)	Prepare *n* empty packs. Wrap the processing handler with a for-loop, and place each packet on the appropriate pack according to its computation result. (e.g., `DirectIPLookup`)

Table 1: Our element conversion method with the varying number of input/output ports.

4 Evaluation

4.1 Experiment Setup

We set up a machine with two quad-core Intel Xeon X5550 (2.66 GHz) CPUs with 1333 MHz 12 GB DDR3 memory. There are two NUMA nodes, one per CPU, and two dual-port Intel 82599 X520-DA2 10GbE cards, one per NUMA node. The maximum possible throughput of this system is 40 Gbps. We use packet generator running on a separate machine with the same number of 10GbE ports that can blast packets at a full line rate of 40 Gbps regardless of the packet size. The four ports in these two machines are directly connected to the corresponding ports.

4.2 Implementation

The system uses the vanilla Linux kernel 2.6.36.4. `psio` driver is configured to use four RX queues (to affinitize to the cores of a CPU in the same NUMA node) and eight TX queues per NIC port. We have modified the codebase of Click 2.0.1 to make it aware of NUMA thread affinity. Our Click configuration specifies an IPv4 router based on an example included in the Click source code, which is extended to use all CPU cores and the `psio` API. We hand-coded the batch-aware elements currently, but we plan to implement automatic conversion of existing elements into batch-aware elements and single-core configurations into multi-core versions.

4.3 Preliminary Results

In our experiment, we measure the throughput and latency by subjecting the system to a workload of 64 B IPv4 packets coming in at a line rate. Figure 2 shows the performance improvements as we apply batching step by step.

The baseline is about 2-3 Gbps using the original Click. Attaching `psio` to unmodified Click shows 7 Gbps. With NUMA-aware thread affinity, the throughput goes up to 16 Gbps, and computation batching (batch processing in intermediate elements) makes 28 Gbps, adding 75% improvement. This is nearly a 10x improvement compared to the original Click. The result is promising in that a little modification to Click can vastly improve the performance.

We find that the latency is also improved with our implementation. When we apply only I/O batching with `psio`, the latency fluctuates between 150 and 200μsec. However, with computation batching, the latency becomes stable with a much lower delay of 20μsec. For underutilized situations with I/O batching only, we suspect that the latency increases or fluctuates due to queuing at the end of the processing path, but the fluctuation disappears when computation batching is applied. We also find that the latency does not increase significantly as the batch size increases. These results indicate that batching has negligible negative impact to the latency and that the overall performance improvement comes with the decreased latency.

For small batch sizes (1 to 8), we observe that computation batching incurs some overhead, but it gets hidden by the performance gain as the batch size increases. From these results, we conclude that the batch size should be larger than 128 to reach a maximum performance on our system.

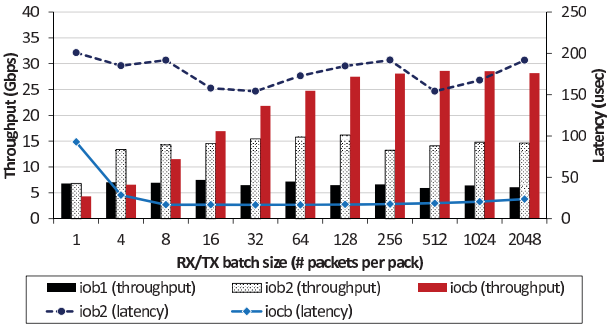


Figure 2: Performance improvements with preliminary implementation of our design. `iob1`: I/O batching only ignoring NUMA thread affinity. `iob2`: I/O batching only. `iocb`: I/O batching + computation batching.

5 Future Work

5.1 Higher Performance

We plan to investigate the I/O performance improvement by introducing pipelining, which allows overlapping of packet reading and processing. The Click modular router currently uses polling to check available input packets from the network. While polling generally decreases the latency, it wastes CPU cycles and increases power consumption even when there are not enough packets to saturate the line capacity. To overcome the shortcomings of polling, we will integrate a hybrid mechanism such as NAPI in Linux [16] by using `psio` and modifying Click's scheduling implementation. In addition, we believe that I/O and processing pipelining can help further. The current implementation of `FromQueue` does not overlap the reading of the RX queues and processing of already-received packets, since `FromQueue` and other elements lie on the same processing path and are executed in serial. To support pipelining, we need an asynchronous I/O scheme that maximizes the CPU utilization.

For higher packet processing throughputs, we consider using many-core processors such as GPUs in favor of larger batch sizes. We know packet computation can be accelerated with GPUs from existing works [10, 11], and we expect that parallelization with many-core processors can curb the

latency increase from batching. The question raised here is: how much performance gain can we get with Click by offloading some elements to GPUs? It is challenging to make real-time decisions on what elements to offload by observing the workload changes inside the processing paths. The difference of computation types that CPUs and GPUs do better makes the scheduling problem more complicated.

5.2 Batch-Split Problem

Deep processing paths may contain many multi-output elements. The pitfall is that different packets are often directed to different destination elements causing a split in a pack, which in turn reduces the batching effect. We plan to analyze how much it impacts the performance, and design an efficient split/merge mechanism of packs to maintain sufficiently large batch size. We expect that splitting a pack into a few smaller batches (1 to 4) would not be a problem if we have large enough RX batch size, but we might need a different strategy for more complex configurations.

6 Conclusion

In this work, we have explored the effect of batching in the context of the Click software router. We find that batch processing in both packet I/O and computation is the key to high performance, and that we should take care of NUMA architectures and multi-queue network cards for multi-core scalability. We have suggested a few simple approaches that extend the existing configurations and elements of Click to be friendly to batch processing. Our evaluation shows that we can improve the performance of Click by almost an order of magnitude with I/O and computation batching. We plan to continue this work by automating the proposed techniques and to explore possibilities to use many-core processors like GPUs.

Acknowledgement

We appreciate valuable feedbacks from anonymous reviewers of APSys. We also thank Jinyoung Jeong for technical assistance and Yoonsung Hong for his suggestions on better writing. This work was supported in part by the KCC (Korea Communications Commission), Korea, under the R&D program supervised by the KCA (Korea Communications Agency) (#08-911-05-002), and by the National Research Foundation of Korea (NRF) grant #2012015222.

References

[1] PF_ring. `http://www.ntop.org/products/pf_ring/`.

[2] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, , and G. Varghese. Endre: An end-system redundancy elimination service for enterprises. In *USENIX NSDI*, 2010.

[3] A. Bianco, R. Birke, D. Bolognesi, J. Finochietto, G. Galante, M. Mellia, M. Prashant, and F. Neri. Click vs. Linux: Two efficient open-source IP network stacks for software routers. In *IEEE HPSR*, 2005.

[4] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for NUMA-aware contention management on multicore systems. In *USENIX ATC*, 2011.

[5] B. Chen and R. Morris. Flexible control of parallelism in a multiprocessor PC router. In *USENIX ATC*, 2001.

[6] M. Dobrescu, K. Argyraki, G. Iannaccone, M. Manesh, and S. Ratnasamy. Controlling parallelism in a multicore software router. In *ACM PRESTO*, 2010.

[7] M. Dobrescu, K. Argyraki, and S. Ratnasamy. Toward predictable performance in software packet-processing platforms. In *NSDI*, 2012.

[8] M. Dobrescu, N. Egi, K. Argyraki, B. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *ACM SOSP*, 2009.

[9] D. Han, A. Anand, F. Dogar, B. Li, H. Lim, M. Machado, A. Mukundan, W. Wu, A. Akella, D. Andersen, J. Byers, S. Seshan, and P. Steenkiste. XIA: Efficient Support for Evolvable Internetworking. In *USENIX NSDI*, 2012.

[10] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *ACM SIGCOMM*, 2010.

[11] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *USENIX NSDI*, 2011.

[12] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 2000.

[13] J. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA–an open platform for gigabit-rate network switching and routing. *IEEE MSE*, 2007.

[14] P. Nikander, B. Nyman, T. Rinta-aho, S. Sahasrabuddhe, and J. Kempf. Towards software-defined silicon: Experiences in compiling Click to NetFPGA. In *1st European NetFPGA Developers Workshop*, 2010.

[15] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC*, 2012.

[16] J. Salim, R. Olsson, and A. Kuznetsov. Beyond softnet. In *Annual Linux Showcase & Conference*, 2001.

[17] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *USENIX NSDI*, 2012.

[18] N. Shah, W. Plishker, K. Ravindran, and K. Keutzer. NP-Click: A Productive Software Development Approach for Network Processors. *IEEE Micro*, 2004.

[19] L. Soares and M. Stumm. Flexsc: Flexible system call scheduling with exception-less system call. In *USENIX NSDI*, 2010.

[20] V. Vasudevan, D. Andersen, and M. Kaminsky. The case for vos: The vector operating system. In *USENIX HotOS*, 2011.