NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors

Joongi Kim Keon Jang Keunhong Lee Sangwook Ma Junhyun Shim Sue Moon

KAIST

{joongi, keonjang, keunhong, sangwook, junhyun}@an.kaist.ac.kr, sbmoon@kaist.edu

Abstract

We present the NBA framework, which extends the architecture of the Click modular router to exploit modern hardware, adapts to different hardware configurations, and reaches close to their maximum performance without manual optimization. NBA takes advantages of existing performance-excavating solutions such as batch processing, NUMA-aware memory management, and receive-side scaling with multi-queue network cards. Its abstraction resembles Click but also hides the details of architecture-specific optimization, batch processing that handles the path diversity of individual packets, CPU/GPU load balancing, and complex hardware resource mappings due to multi-core CPUs and multi-queue network cards. We have implemented four sample applications: an IPv4 and an IPv6 router, an IPsec encryption gateway, and an intrusion detection system (IDS) with Aho-Corasik and regular expression matching. The IPv4/IPv6 router performance reaches the line rate on a commodity 80 Gbps machine, and the performances of the IPsec gateway and the IDS reaches above 30 Gbps. We also show that our adaptive CPU/GPU load balancer reaches near-optimal throughput in various combinations of sample applications and traffic conditions.

1 Introduction

High-performance commodity hardware has enabled emergence of software packet processing systems that extract close to maximum performance out of the given hardware [3, 5, 21, 33, 44, 54]. Key hardware enablers include multi-core CPUs, mass-market many-core processors such as GPUs, and cheap 10 GbE network cards. The key techniques in software to exploit hardware performance include batching, pipelining, and parallelization.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroSys'15, April 21–24, 2015, Bordeaux, France.
Copyright © 2015 ACM 978-1-4503-3238-5/15/04…$15.00.
http://dx.doi.org/10.1145/2741948.2741969

The challenge is that implementing and tuning such a complex mix is costly. Combining existing techniques requires architecture-specific expertise for application developers, which is difficult to generalize for diverse hardware and applications. As a result, many systems resort to reinventing the wheel due to a lack of general frameworks or reusable libraries. The current best practice for optimization is ad-hoc and manual, as well. Searching the space of inter-dependent tuning parameters such as the batch size and the processor load balancing ratio is time-consuming and configuration-dependent. The diversity in hardware and software, such as the application behaviors, heterogeneous processor architectures, and dynamically changing workloads, often result in suboptimal performance.

We argue that existing work should become a reusable building block for future systems. Historically, the Click modular router has laid the foundation for a programmable router framework [31], and follow-up work has improved Click's performance. DoubleClick [29] has demonstrated the potential of computation batching. Snap [47] adds GPU-offloading abstractions to Click.

In this work we propose a software-based packet processing framework called *NBA (Network Balancing Act)*. It exploits the latest hardware advances, but encapsulates their low-level specifics. It provides application developers with a familiar programming model that follows the Click modular router, while it achieves close to maximum application performance. This way, developers can focus on the application logic and leave the architecture-specific tuning to the framework.

The key contributions of this work are the following designs in NBA:

- A batch-oriented modular architecture with minimal performance overheads by applying efficient memory management and branch prediction,
- A declarative abstraction for GPU offloading that reduces the learning cost for application developers and eases implementation effort,

	Criteria	Click Modular Router [31]	RouteBricks [15]	PacketShader [21]	DoubleClick [29]	Snap [47]	**NBA**
Established Techniques	IO Batching	○ (with netmap)	○	○	○	○	○
	Modular Programming Interface	○	○	×	○	○	○
Our Contributions	Computation Batching	×	×	△ (no branches)	△ (manual)	△ (partial)	○
	Declarative Offloading Abstraction	×	×	△ (monolithic)	×	△ (procedural abstraction)	○
	Adaptive Load Balancing for Heterogeneous Processors	×	×	×	×	×	○

Table 1: Comparison of existing packet processing frameworks and NBA.

- An adaptive CPU/GPU load balancing algorithm that eliminates optimization effort by finding the maximum throughput under any mix of workloads.

Our implementation of NBA reaches up to 80 Gbps for IP routing and above-30 Gbps for IPsec encryption and pattern-matching intrusion detection system (IDS) on a single commodity machine comprised of dual Intel Sandy Bridge CPUs, two desktop-class NVIDIA GPUs, and eight 10 GbE ports, under the hardware budget $7,000[1]. We also show general applicability of NBA's adaptive load balancer by comparing its performance on multiple combinations of application and traffic conditions.

The rest of the paper is organized as follows. In the next section (§ 2) we introduce the requirements and technical challenges. § 3 describes concrete design choices and the implementation details. We evaluate our framework in § 4 and introduce related work in § 5, followed by discussion in § 6, future work in § 7, and finally conclusion in § 8.

2 Motivations and Challenges

The requirement of high performance and programmability poses the following recurring challenges in packet processing frameworks:

- Achieving performance scalability on multi-core/multi-socket systems
- Providing an abstraction for packet processing to ease adding, removing, and changing the processing functions
- Reducing overheads of frequent operations, particularly per-packet function calls and resource allocation
- Handling the complexity of offloading computations to accelerators such as GPUs

In Table 1, we summarize how existing body of work has addressed the above challenges. For example, batch processing for packet IO and computation is the essential method to reduce per-packet overheads. As all work facilitates some form of batch processing for packet IO (IO batching), we conclude it has become the intrinsic part of packet processing frameworks. For the abstractions, that of the Click modular router has been reused over time in nearly every work, showing versatility and popularity of Click's abstraction. The missing pieces are batch processing in packet processing pipelines (computation batching) and facilities to ease the complexity of offloading, such as an easy-to-use offloading abstraction and adaptive load balancing. These are where NBA comes in.

Below we investigate why the mentioned challenges are important and overview our solutions.

Multi-core and multi-socket scalability

Scalability in packet processing frameworks has become an intrinsic requirement as high line rates per port ($\geq$ 10 Gbps) are becoming common [12]. The current commodity servers have two fundamental limits that mandate scalable software designs: *i*) the PCIe and memory bandwidth limit which necessitates scalability on NUMA (non-uniform memory access) multi-socket systems, and *ii*) the clock speed limit which necessitates scalability on multi-core CPUs. Prior work has already shown that both types of scalability are important: *i*) using remote sockets' memory increases the packet processing latency by 40-50% and reduces the throughput by 20-30% [21], and *ii*) exploiting all cores is essential to reach beyond 10 Gbps line rates [5, 15, 16, 18, 21, 27, 29, 44].

Following prior work, NBA carefully chooses and embraces known techniques for high scalability. We describe the details in § 3.1 and § 3.2.

Packet processing abstraction

We reuse and extend the Click modular router's abstraction [31] for NBA. In Click, the packet processing operators are represented as *elements* that generate, process, or discard packet objects. The elements are written as C++ classes with a standard packet manipulation interface including raw buffer access so that they can implement any kind of new packet operators. It composes elements into a directed graph using a declarative composition language, which exposes the pipelining structure. NBA extends Click's abstraction by implementing batch processing of elements and adding packet-level parallelization to each element by supporting accelerators.

We believe that Click will continue to be a dominant abstraction due to its extensive set of reusable elements that

[1] All prices are from `http://amazon.com` in October 2014.

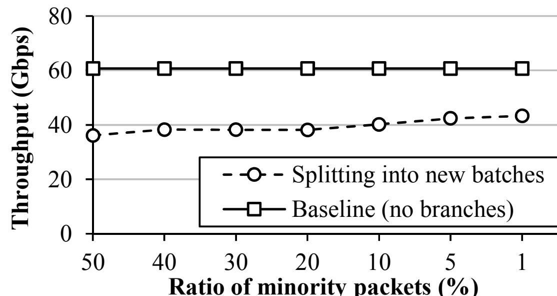


Figure 1: Throughput drops by the relative size of split batch (the smaller one among two split batches).

come with the easy-to-use composition language and C++'s versatility on writing new elements.

Reducing per-packet overheads

The number of minimum-sized packets in a 10 Gbps link is over 14 millions per second. The CPU cycle budget for each packet is less than 200 cycles with a single CPU core running at 2.67 GHz. This means that any per-packet operation cost must be minimized and amortized via batch processing. Prior work has already demonstrated the effectiveness of batch processing in both packet IO [3, 5, 6, 44] and computation [21, 29].

However, a new challenge arises in combination of computation batching and the fine-grained element abstraction from Click. Since each element may have multiple output edges and individual packets in a batch may take different edges (processing paths), the batch needs to be reorganized after such branches. The challenge here is to avoid such batch reorganization overheads. We take an approach similar to [29]: split the batch into smaller batches where each split batch has packets taking the same path only so that later elements do not have to check the applicability of operations packet by packet. However, splitting batches causes two types of performance impacts: allocation overheads for new batch objects and decreased batch sizes. Figure 1 demonstrates the performance degradation by batch splits, up to 40%. The result suggests that the primary overhead (25%) comes from memory management, allocating new batches and releasing the old batch, since the split batch sizes impact the throughput within 15%.

We tackle this issue in two ways: (*i*) avoiding branches with multiple output edges and (*ii*) devising a simple batch-level branch prediction technique to reduce the memory management overheads. We describe the details in § 3.2.

Complexity of exploiting accelerators

A series of prior work have shown the potential of GPUs as packet processing accelerators [21, 26, 27, 47, 52]. Accelerators compliment the computation power of CPUs with their specialization to exploit data parallelism. Examples of accelerators range from general-purpose GPUs to many-core processors such as Intel Xeon Phi coprocessor and Tilera. They invest most of their silicon budget to a massive number of arithmetic processing cores (a few tens to thousands) to

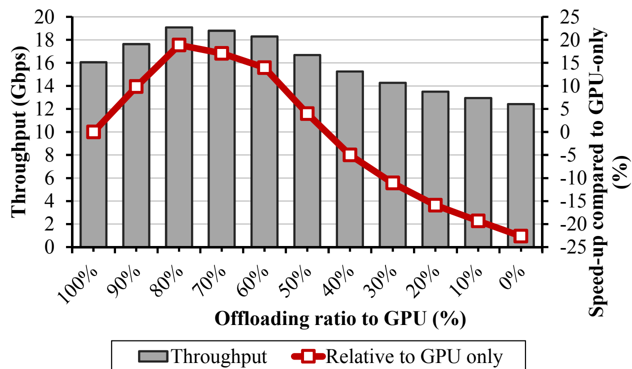


Figure 2: Performance variations of the IPsec encryption gateway with varying offloading fractions. The offered workload is a packet trace from CAIDA 2013 July dataset.

exploit data parallelism, in contrast to desktop/server CPUs that have a small number of fat cores (four to eight typically) with large caches to run individual threads faster.

We have two challenges to fully exploit such accelerators. First, it is difficult and time-consuming to write codes that cover GPU offloading specifics and run fast by carefully handling vendor-specific details. Second, offloading everything always may not yield the best achievable performance.

Offloading abstraction: The major hassle to handle accelerators is data copy and synchronization between the host and accelerator device with concerns on vendor-specific details. For example, GPU networking solutions [26, 27, 51] have used multiplexed command queues to exploit pipelining opportunities in data copies and kernel execution. When doing so it is performance-critical to avoid implicit synchronization of command queues, but unfortunately it is easy to overlook vendor-specific caveats. One such example is `cudaStreamAddCallback()` in CUDA [2]. Its intention is to add a completion notification point to a command queue which asynchronously notifies the caller thread via a callback function, but it actually synchronizes with all ongoing requests in other queues when invoked. To this end, we argue that existing frameworks expose too much of such details to application developers. PacketShader [21] offers a simple interface to write packet processing applications using preprocessing, computing, and postprocessing callbacks. Inside them, application developers are forced to write codes that deal with all GPU offloading specifics such as buffer management, pipelining, and synchronization by themselves. Snap [47] is the first to have offloading abstractions as composable Click modules, but it remains in a procedural abstraction where application developers must understand, specify, and optimize the order of offloading steps such as data copies and kernel execution.

Load balancing: Offloading computations for all incoming packets may not yield the best performance; we need to find the optimal balance between the CPU and accelerators. Offloading requires preprocessing and postprocessing

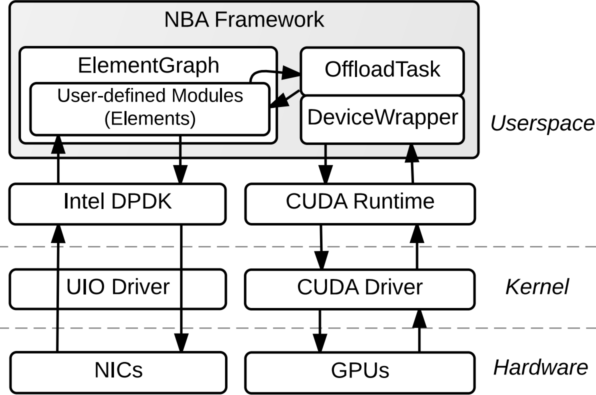


Figure 3: The high-level software architecture.

steps and has inevitable overheads due to them. If not used judiciously, offloading may hurt the performance instead of improving it. For example, our experiment using a sample implementation of IPsec encryption gateway shows a corner case where offloading all workloads does not yield the optimal performance. Encryption is a compute-intensive task and offloading it is expected to yield better performance, but it does not always. Figure 2 illustrates the performance variation by the fraction of offloading. Here the offloading fraction 30% means that 70% of input packets are processed by the CPU and 30% by the GPU. The packets to offload are selected randomly with the probability 0.3. The result shows that the maximum performance is achieved when we offload 80% of traffic to GPUs, yielding 20% more throughput compared to GPU-only and 40% more than CPU-only settings.

We explain the structure of offloadable elements with a suggested declarative offloading I/O abstraction in § 3.3 and describe our adaptive load balancing scheme in § 3.4.

3 Design and Implementation

In this section we describe details on how we address the challenges in the framework design and implementation. Our primary concern is to hide the details of batch processing and offloading from application developers. As Figure 3 shows, we implement our framework on top of Intel DPDK and NVIDIA CUDA runtime.

3.1 Packet IO Layer

The packet IO layer is the most performance-sensitive component in the framework as it decides the available budget for packet processing. As prior work has reported [21, 33, 54], passing through the Linux kernel network stack impairs the raw packet IO performance due to unnecessary protocol handling and memory management overheads inside the kernel. As NBA itself becomes a base for customized packet processing applications, we need to minimize the overheads between NIC and NBA. For this purpose, we have a number of choices [3, 5, 6, 44] that offer high-performance user-level packet IO schemes suitable for NBA.

Among them we choose Intel DPDK [3] because it does not only have zero-copy packet IO APIs but also provides

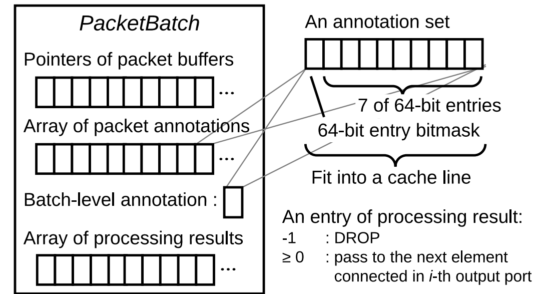


Figure 4: The structure of a packet batch.

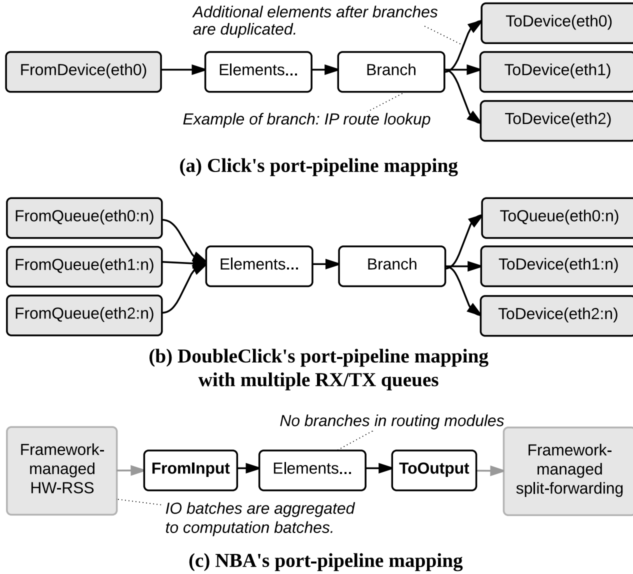
a comprehensive NUMA-aware, architecturally optimized memory management libraries that ease development of a multi-socket/multi-core scalable framework. For example, its memory pool library is particularly useful for handling the path diversity of packets in a modular pipeline where we need to allocate and release individual packet buffers at different times with minimal overheads. It provides other useful utilities as well, such as lock-free rings and thread management libraries that ease development of a high-performance packet processing framework. Yet another advantage of DPDK is participation of NIC vendors in its development process, which allows us to test latest NIC models in a timely manner. Nonetheless, NBA itself is not fundamentally limited to work with DPDK only, because NBA is not tied with DPDK's internals. Adding a wrapper to the DPDK APIs will be sufficient to replace DPDK with other user-level packet IO libraries.

3.2 Batch-oriented Modular Pipeline

Packet batches as first-class objects

On top of the packet IO layer, NBA wraps received packets into *packet batches* for computation batching and feeds them into the modular pipeline (`ElementGraph` in Figure 3). Similarly to Click, the `ElementGraph` traverses user-defined modules (*elements*) in the pipeline until an element claims to store the batch or all its packets are dropped or transmitted out. Each element defines a reusable packet processing function. Although NBA's programming model follows Click, we use packet batches as our universal input/output object type for elements instead of individual packets.

We design packet batches to be a light-weight, efficient data structure as our system should cope with 10K to 100K of packet batches per second. To avoid excessive copy overheads when creating new batches, packet batches do not carry actual packet contents but only the pointers to packet buffers. Hence, its structure is a simple set of arrays as shown in Figure 4: an array of pointers to packet buffers, an array of per-packet processing results including the output link IDs to next elements, a per-batch annotation set, and an array of per-packet annotation sets. We use annotations to attach extra metadata to packets (*e.g.*, timestamp and input NIC port index) and allow data sharing between different elements (*e.g.*, flow IDs for protocol handling). The commonly used



(a) Click's port-pipeline mapping

(b) DoubleClick's port-pipeline mapping with multiple RX/TX queues

(c) NBA's port-pipeline mapping

Figure 5: How NBA avoids multi-edge branches.

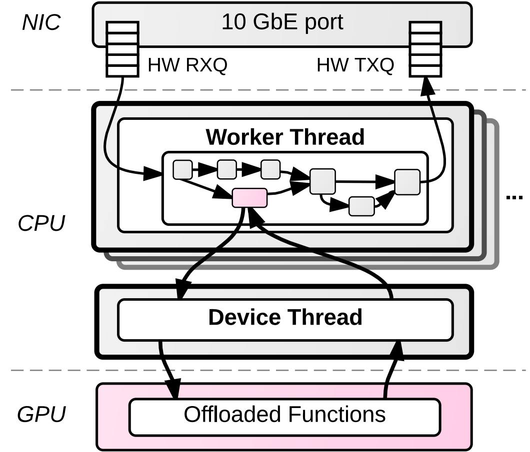


Figure 6: An example of thread and core mapping on a system with a single socket quad-core CPU and one GPU running three worker threads and one device thread. Additional CPU cores in a socket add more worker threads.

annotation fields are restricted to 7 entries to make the annotation fit into a cache line for performance.

Hiding computation batching

NBA runs an iteration loop over packets in the input batch at every element whereas elements expose only a *per-packet* function interface. The framework handles the processing results of individual packets (*e.g.*, drop or send it to a specific next element) by splitting packet batches when the element is a branch, *i.e.*, has multiple next-hop elements.

In addition to per-packet elements, we introduce *per-batch* elements as well to run coarse-grained operations efficiently. For example, making load balancing decisions in packet granularity incurs high overheads and coarse-grained decisions still work since we have a very high packet input rates, millions per second. Another example is a queue element because storing packets can be done in the unit of incoming batches "as-is" without decomposing them. NBA keeps universal composability of both per-packet and per-batch elements as it decides whether to use an iteration loop or direct function call depending on the type of element.

NBA takes advantage of the Click configuration language to compose its elements, with a minor syntax modification to ease parsing element configuration parameters by forcing quotation marks around them. We have plans to extend and clarify our modified Click syntax [2].

Dealing with the batch split problem

Per-packet semantic of elements raises the batch split problem with multiple output edges, or branches, as we discussed in § 2. NBA tackles it in two ways: avoiding branches with multiple output edges that originate from system resource mappings and use of a branch prediction technique to reuse the batch object for the processing path that most packets take.

To reduce the occurrence of such multi-edge branches, we separate out the hardware resource mappings from elements. For example, Click's IO elements such as `ToDevice` and meta-elements such as `StaticThreadSched` represent not only the functionality but also hardware resources. This coupling causes configurations to have multi-edge branches (*e.g.*, `IPLookup`) to split the traffic into multiple resources (*e.g.*, multiple outgoing NIC ports). As Figure 5 illustrates, NBA moves the hardware resource mapping and the split-forwarding logic into the framework to remove multi-edge branches where batches are split into similar-sized batches. Routing elements now use annotation to specify the outgoing NIC port and the framework recognizes it after the end of the pipeline. It allows us to simply drop invalid packets so that we have no branches at all in the configurations used in this paper.

With the help of removing multi-edge branches, we apply a simple branch prediction technique because (*i*) most branches have only two edges and (*ii*) most packets take one path and only few exceptional packets (*e.g.*, invalid ones) take the other path after such branches. A typical case is the `CheckIPHeader` element. The branch prediction works as follows. Each output port of a module tracks the number of packets who take the path starting with it, and the framework reuses the input packet batch object for the output port where the largest number of packets has passed last time. In the reused packet batch, dropped packets and packets that have left are masked out instead of shrinking the pointer/annotation arrays to avoid extra overheads.

Multi-core scalable pipelines

NBA uses the replicated pipeline model combined with RSS. As [16, 18] analyze and [26] confirms, this model minimizes cache bounces caused by moving packets core to

[2] We have open-sourced our parser as a separate library at `https://github.com/leeopop/click-parser`

core and synchronization overheads. NBA has two types of threads: worker threads and device threads.

Worker threads run the replicated pipelines (with replicated instances of `ElementGraph`) following the run-to-completion processing model offered by Intel DPDK[3]. They run IO loops that synchronously fetch received packets from NICs, process them, and transmit out or discard. The IO loop also checks offload completion callbacks from the device thread. The other possible model is pipelining, at which we separate IO and computation threads (and cores): the IO threads enqueue the received packets into a ring shared with computation threads that asynchronously process the packets. In early stages of NBA development, we have tested the pipelining model with and without hyperthreading but the performance was lower than the run-to-completion model, as also reported by [26]. Worker threads also follow the shared-nothing parallelization model; there is no synchronization between worker threads at all because nothing is shared. However, to reduce cache misses, we allow sharing of large read-dominant data structures such as forwarding tables via a node-local storage by which elements can define and access a shared memory buffer using unique names and optional read-write locks.

Device threads manage communications with accelerators such as GPUs. As worker threads send offload tasks containing packet batches and element information, they execute them on a pool of command queues for the configured accelerators. There is one device thread per NUMA node per device, assuming all NUMA nodes have the same set of offload devices.

Putting them together, Figure 6 illustrates NBA's thread and core mappings. We provide a scripting interface to allow customized mapping of system resources and NBA's software components such as threads and queues between worker and device threads.

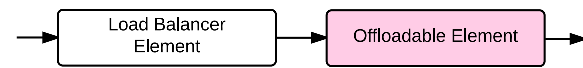
Scheduling of elements

As the result of tailoring Click's modular abstractions for the run-to-completion model, NBA unifies its separate push/pull processing semantics into push-only processing.

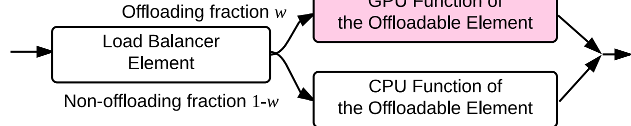
Like Click [13], NBA has schedulable elements where the processing begins. But differently, the packet output element (`ToOutput`) are not schedulable as we transmit packets synchronously and no queue is required by default. Schedulable elements have a special method named `dispatch()` which the framework executes on every iteration of the IO loop. The element optionally returns a packet batch object to continue processing with descendant elements. It can also set the delay until next invocation to make a timer. `FromInput` element is a typical example of schedulable elements, as it returns the packet batch by querying the framework on every IO iteration.

[3] Section 8 Poll Mode Driver, Intel DPDK Programmer's Guide, from `http://intel.ly/1vVKc1D` accessed at Oct 10, 2014.

Application developer & pipeline configuration's view:



Load balancer's view:



How NBA executes:

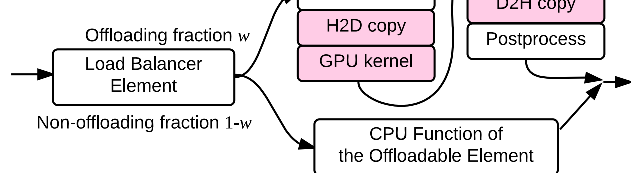


Figure 7: How NBA interprets an offloading element with a load balancer element. The shaded parts are executed in device threads and accelerators (GPUs).

3.3 Offloading to Accelerators

Offloadable elements and their interpretation

Offloadable elements define a CPU-side function and an accelerator-side function and its input/output data formats. As Figure 7 shows, if the load balancer decides to offload the input batch, NBA automatically handles the process of running the accelerator-side functions including preprocessing of the input data, host-to-device data copies, kernel execution, device-to-host data copies, and postprocessing of the output data.

Declarative input/output formats

We suggest a declarative abstraction to express input/output data definitions: *datablocks*. Using a declarative abstraction has two advantages: safety guarantee and automatic optimization. Safety means that the framework validates packet data and datablock fields prior to execution. It allows application developers to forget buffer management details, which is bug-prone and time-consuming, by specifying what packet processing functions uses what data only. Declarative abstraction creates the room for automatic optimization as well. The framework can analyze the datablock fields and extract chances of reusing GPU-resident data between different offloadable elements or coalescing copies of different datablocks. Individual elements are still independent and unaware of other elements accessing the same datablocks or not, but the framework can reuse datablocks by calculating the lifecycle of datablocks and delay postprocessing until all offloadable elements finish using relevant datablocks.

Datablocks contain the input/output byte ranges in packets and/or user-defined preprocessing/postprocessing functions as shown in Table 2. Each datablock is mapped to a page-locked memory buffer for host-to-device and/or

IO Type	Fields
partial_pkt	offset, length, alignment
whole_pkt	offset, alignment, size-delta
user	length, pre/postproc func.

Table 2: Datablock format information.

device-to-host data copies depending on their I/O semantics. `partial_pkt` and `whole_pkt` copies input/output data from/to packet buffers and `user` lets a user-defined function take the packet batch and write or read what it wants.

In this paper, we leave the full implementation and suggested optimization as future work. The implementation used in the evaluation mandates all offloadable elements to implement I/O buffer management and pre-/post-processing steps manually and monolithically. At the time of writing, our on-going implementation has 10 to 30% performance overheads with datablock abstraction, but we expect that the automated optimization techniques and further framework optimization would reduce the overhead.

How offloading works

We use NVIDA's GPU and CUDA to implement acceleration in this paper. NBA offers a shim layer that resembles the OpenCL API [46], and it is easy to extend to support other accelerators such as Intel Xeon Phi or AMD's GPU.

Exploiting parallelism is essential to achieve maximum utilization of GPUs. The batch size requirement for maximum throughput can be as large as thousands of packets [21], which is much larger than the IO and computation batch size. Simply increasing the computation batch size leads to problems in CPU processing performance due to increased cache misses. To solve this, NBA aggregates multiple packet batches just before accelerator offloading. We find that approximately thousands of packets are enough for all the workload in this paper, and set the maximum aggregate size to 32 batches.

3.4 Adaptive CPU/GPU Load Balancing

To minimize manual performance optimization, we design a simple adaptive load balancer. Our focus is to avoid specialization to specific applications or hardware and to find an optimal offloading fraction that yields the maximum performance in any combination of them. Without any prior knowledge or assumption of the environment, adaptive (or feedback-controlled) load balancing is the way to go because the only information available for load balancing decision is the history system states.

How load balancing works

We implement our load balancer as elements to allow application developers to easily replace the load balancing algorithm as needed. A load balancer element chooses the processor of packet batches before they enter the offloadable elements as illustrated in Figure 7. The load balancing decision is stored as a batch-level annotation indicating the index of available computation devices. NBA reads this value and offloads the batch to the designated device, the GPU in our setup. If it is not set, NBA executes the CPU-side function like non-offloadable modules. We expose a system inspector interface to load balancer modules to help their load balancing decision and give feedbacks. The system states include a variety of statistics such as the number of packets/batches processed after startup. Using our batch-level or packet-level annotation support, the users can even add new load balancing targets instead of selection of CPU/GPU, *e.g.*, load balancing between multiple output NIC ports.

Balancing target and algorithm

We devise a load balancing algorithm that maximizes the system throughput. A few existing approaches have devised algorithms with the same goal, namely, opportunistic offloading [27] and dynamic offloading [26]. These two algorithms observe the input queue length and choose to use GPUs when the length exceeds a certain threshold. Dynamic offloading is an advanced version of opportunistic offloading as it has buffers to absorb small fluctuations of the queue length when changing the processor.

Unfortunately, we cannot directly use their algorithms because we have no input queues due to the run-to-completion processing model, at which we process packets in the speed that the CPU can sustain. Even if it is possible, we need to manually tune optimal thresholds for our system and for each application, and that is what we want to avoid.

Avoiding such complexities, we let our load balancer to observe the system throughput directly, squashing all the implementation-specific and application-specific details into a black-box. It chooses the direction (increment or decrement) of the offloading fraction $w \in [0\%, 100\%]$ by δ observing if throughput increases or decreases. The throughput is measured by the number of packets transmitted out per 10K CPU cycles. To avoid being trapped inside local jitter, we use the moving average of throughput and let the load balancer wait for all worker threads to apply the updated fraction values before next observation and update.

By trial and error, we find the generally applicable parameters: how much δ should be, how big history we need to observe, and how long the update interval should be. It is sufficient to smooth out jitter and converge by setting the moving average history size of w to 16384, δ to 4%, and the update interval to 0.2 second. Considering that the arrival rate of packet batches ranges from 10K to 220K in our settings depending on the workloads and the history size corresponds to about 0.1 to 1.5 seconds. We also gradually increase the waiting interval from 2 to 32 update intervals when we increase w from 0 to 100% as higher w incurs jitter persisting for a longer period of time. We continuously insert perturbations (same to δ) to the value of w, to allow it to find new a convergence point when the workload changes.

Category	Specification
CPU	2x Intel Xeon E5-2670 (Sandy Bridge) (octa-core 2.6 GHz, 20 MB L3 cache)
RAM	32 GB (DDR3 1,600 MHz 4GB x8)
NIC	4x Intel X520-DA2 (dual-port 10 GbE, total 80 Gbps)
GPU	2x NVIDIA GTX 680 (1536 CUDA cores, RAM 192 GB/s, PCIe 3.0)

Table 3: Hardware configuration

3.5 Implementation Efforts

NBA consists of 24K lines of C++ and CUDA code, excluding configurations and micro-benchmark scripts[4]. It took almost a full year from scratch to a working version including the adaptive load balancer.

4 Evaluation

4.1 Experiment Setup

Hardware and software: Table 3 lists the specification of the machine we used. All machines have the same Ubuntu Linux 14.04.1 with unmodified Linux kernel, NVIDIA CUDA 6.5, and Intel DPDK 1.7.0.

Notation: When we say "CPU-only" in the following experiments, it means all elements including offloadable elements are processed using their CPU-side functions only. "GPU-only" means all offloadable elements are processed using their GPU-side functions only, but all other elements are still processed by the CPU.

Sample applications: We choose four sample applications, IPv4 router ("IPv4"), IPv6 router ("IPv6"), IPsec encryption gateway ("IPsec"), and IDS, that have various performance characteristics to show NBA's general applicability. IPv4 is memory-intensive as it performs at most two table lookups with bit-masking IP destination addresses. IPv6 is both memory and computation-intensive as it performs binary search for every destination address over a large table. IPsec is highly computation-intensive as it performs encryption and hashing, but is also IO-intensive because it needs to copy the packet payloads from/to the GPUs. IDS is compute-intensive as it performs pattern matching and also IO-intensive like IPsec, but with only host-to-device copies.

Figure 8 shows our sample routing applications expressed in pipeline configurations. IP lookup elements are based on PacketShader's implementation, used under authors' grant [21], using DIR-24-8 algorithm for IPv4 [20] and binary search for IPv6 [53]. The IPv4 lookup algorithm performs two random memory accesses while the IPv6 lookup algorithms performs at most seven random memory accesses. IPsec encryption uses HMAC-SHA1 to authenticate the packets and AES-128CTR to encrypt them. We implement it to exploit AES-NI for faster computation AES in recent CPU models. To enable AES-NI for OpenSSL in the CPU version, we use its envelope API but with a trick: initialize envelope contexts for all flows on startup and reuse them by only changing initial vector (IV) values to avoid memory management overheads in the data-path. Otherwise we see context initialization overheads overshadow the performance benefit of AES-NI. Our IDS uses Aho-Corasik algorithm for signature matching and PCRE for regular expression matching [4, 8] with their DFA forms using standard approaches [48].

[4] Measured using CLOC. See `http://cloc.sourceforge.net/`.

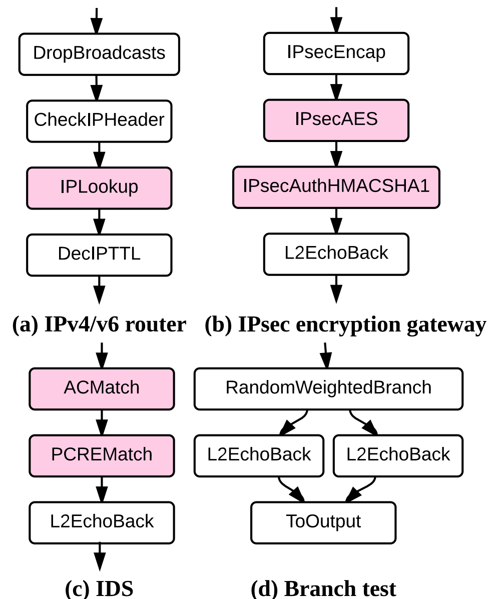


Figure 8: The configurations of our sample applications. The highlighted elements are offloadable. We omitted common `FromInput` and `ToOutput` elements for space.

Workloads: Unless otherwise specified, we use a randomly generated IP traffic with UDP payloads and offer 40 Gbps load from two separate packet generator machines, 80 Gbps in total. For IPv6 router application, we use IPv6 headers and IPv4 headers for other cases.

4.2 Minimizing Abstraction Overhead

Computation batching: Figure 9 shows how helpful computation batching is in small packet sizes. IPv4/IPv6 routers and IPsec encryption gateway with 64 B packets shows significant performance improvements, by 1.7 to 5.2 times. With large packets, we see IP routing applications reach 80 Gbps regardless of batch sizes but IPsec has about 10% of performance degradation with no computation batching.

Among various combinations of IO and computation batch sizes, we set the default IO and computation batch sizes to 64 packets and the offloading batch size to 32 packet batches as described in § 3.3. In all sample applications, this default IO and computation batch sizes give 97 to 100% of the maximum throughput found by manually searching different combinations of IO and computation batch sizes using minimum-sized packets. The default offloading batch size yields 92 to 100% as well compared to manually found optimal values. Hence, all experiments use the same setting.

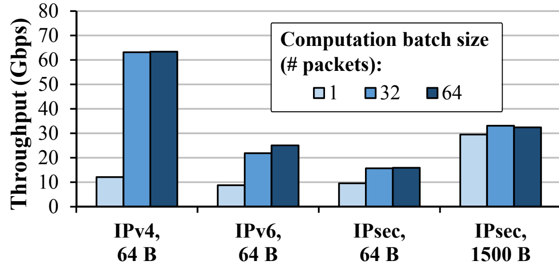


Figure 9: Performance improvements by computation batching.

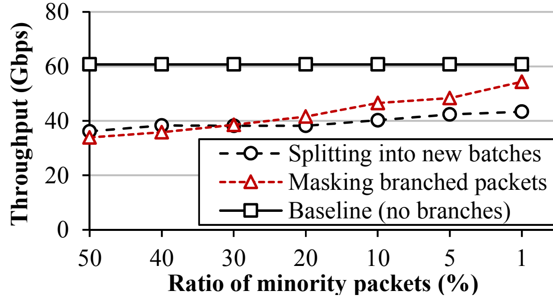


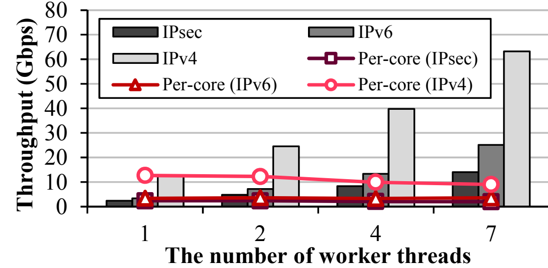
Figure 10: Performance benefit of our branch prediction technique compared to the worst case in Figure 1.

Composition overhead: We evaluate the overhead of passing multiple modules using a linear pipeline configuration with many no-op elements. We measure the latencies of a linear path consisting of multiple no-op elements without any computation. To see the minimal processing time except extra queuing delays in the switch and NICs, we offer 1 Gbps traffic. The baseline latency with zero no-op elements and minimal L2 forwarding is 16.1 μsec on average. Adding no-op elements increases the latency, but only by about 1 μsec after adding 9 no-op elements. This result indicates that the major performance impacting factor for a linear pipeline is the computation complexity, not the number of elements.

Gains of branch prediction: Next we evaluate the gains of branch prediction using a synthetic configuration that has two paths after a branch in Figure 8(d). In Figure 10 we compare three cases: *i*) baseline that simply echoes back all packets without any branch (the solid black line), *ii*) a worst case that put all packets into new split batches (the dotted red line), and *iii*) our branch prediction case that reuses the input batch for the majority of packets and masks their slots in the reused batch (the dotted black line). The worst case degrades the performance 38 to 41% due to excessive memory/pointer copies and allocation of new packet-batch objects. Our branch prediction also has overheads but limiting the degradation to 10% when 99% of packets remain in the reused batch. It also shows that linearization of pipelines is critical to achieve high performance with branch prediction, to decrease the amount of minority packets.

4.3 Multicore Scalability

In this experiment we show how scalable NBA is by using different numbers of worker threads. Figure 11 shows



(a) Scalability of CPU-only throughputs

(b) Scalability of GPU-only throughputs

Figure 11: Performance variation of applications depending on the number of worker threads, using either CPUs only or GPUs only. Note that the last CPU core is dedicated for the device thread, limiting the maximum number of worker threads to 7.

that NBA has marginal overheads when the number of cores increases to more than 4, and the GPU-only cases have more overheads than the CPU-only cases. The reason is that a single dedicated device thread handles all offload tasks from the worker threads in the same CPU, and this incurs synchronization overhead of task input queues. We also observe that the CUDA runtime has significant internal locking overhead by profiling, which consumes from 20% (in IPv4 router with 1 worker threads) to 30% (in IPsec encryption with 7 worker threads) of CPU cycles in cores where the device threads run and the CUDA runtime implicitly spawns its own child threads to poll the device states. We suspect that this is due to our excessive calls of `cudaStreamQuery()` to check if device-to-host copies are finished, but we do so because its performance was better than other methods such as `cudaStreamAddCallback()` or waiting for event objects in the command queue. Nonetheless, we believe that this overhead is specific to CUDA's implementation, not a fundamental problem in other types of accelerators as well.

4.4 Variability of Optimal Points

Figure 12 shows that the optimal balance between CPUs and GPUs highly depends on the workloads including both application type and packet sizes.

For each sample application, we vary the packet size from 64 to 1500 bytes and configure the system to run in extreme conditions: either CPU-only or GPU-only mode. Overall, the IPv4 router and IPv6 router shows increasing throughputs as the packet size increases, reaching up to 80 Gbps. IPsec

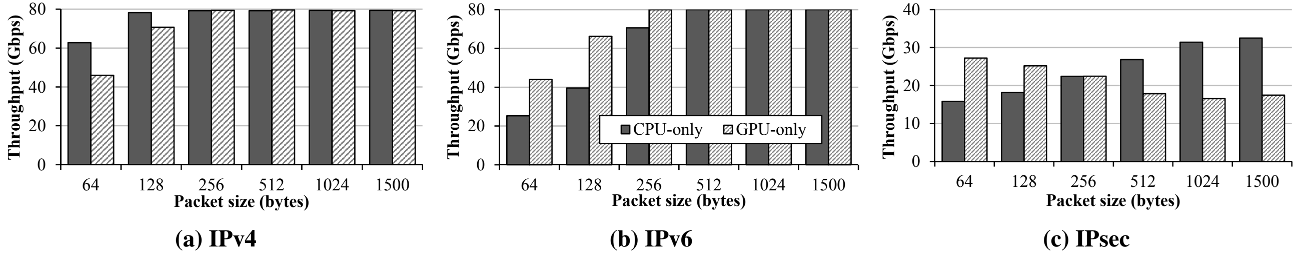


Figure 12: NBA's application performance depending on packet sizes, showing necessity of CPU/GPU load balancing.

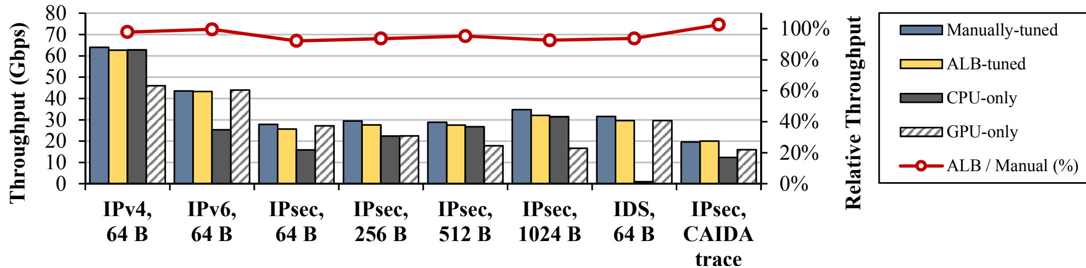


Figure 13: The performance of our adaptive load balancer under several types of workloads.

encryption gateway achieves 23 to 33 Gbps and IDS up to 35 Gbps. In IPv4 router, the CPU performs always better (0 to 37%) in contrast to the IPv6 router where the GPU performs always better (0 to 75%). In IPsec, the GPU performs better (up to 68%) in small packet sizes ($<$ 256 bytes) but this is reversed in large packet sizes as the GPU performs almost twice better. The IDS has significant performance boosts by GPU acceleration, showing from 6x to 47x throughput improvments compared to its CPU-only cases.

As the result indicates, IP routing applications and IDS have consistent trends that either CPU or GPU performs better regardless of packet size, but IPsec encryption gateway shows that the optimal processor may be different depending on packet size and traffic conditions.

4.5 Adaptive Load Balancing

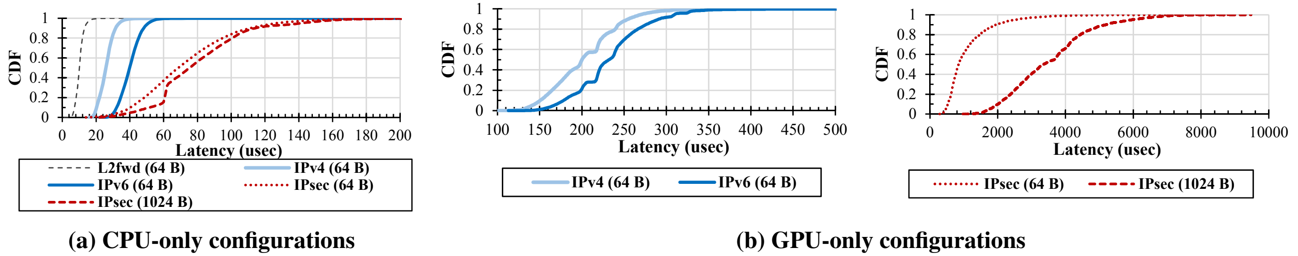
Figure 13 shows the performance of our adaptive load balancer (ALB), compared with the CPU-only/GPU-only cases and manually tuned throughput with exhaustive searches on the offloading fractions. The cases are categorized by the pair of application type and traffic type (randomly generated packets with fixed sizes or a trace replay). We have chosen the representative cases from Figure 12 where the CPU performs better (IPv4 64 B, IPsec 512 B, IPsec 1024 B), the GPU performs better (IPv6 64 B, IPsec 64 B, and IDS 64 B), or mixing them performs better (IPsec 256 B and IPsec with CAIDA trace as in Figure 2).

In all cases, ALB achieves more than 92% of the maximum possible throughput. The particular cases of IPsec 256 B and IPsec with CAIDA traces show the necessity of ALB, as either using the CPU or GPU only does not yield the maximum throughput. Other cases prove that at least ALB does not perform worse than a dumb balancing to use either CPU or GPU only.

4.6 Latency

Figure 14 illustrates the latency distribution of NBA with CPU-only and GPU-only configurations. We have offered medium-level workloads that can be processed without packet drops, and measured round-trip latency using timestamped packets. The latency of the L2 forwarder (L2fwd in Figure 14a) shows the minimum latency of the NBA framework, where 99.9% of packets return within 43 μsec. The L2 forwarder is composed of a single element that transmits the packets in a round-robin fashion using all NICs after exchanging the source and destination MAC addresses. In the CPU-only configurations of IPv4/IPv6 routers, 99.9% of packets return within 60 μsec and IPsec-encrypted packets return within 250 μsec. For each configuration we used the optimal batch sizes, *e.g.*, used large batches (128 packets) in L2fwd and small batches (64 packets) in others. The GPU-only configurations have higher average latency than the CPU-only ones, about 8 to 14$\times$, with higher variances. The large gap between 64 B and 1024 B cases in IPsec exhibits the data copy overhead for offloading to GPUs.

Overall, NBA's minimum and average latency come close to the state-of-the-art. Snap [47] reports the minimum latency of its CPU-only and GPU-only configurations of an IDS router as 31.4 μsec and 292 μsec on average respectively, where they are composed of radix-tree lookup and Aho-Corasick string matching algorithms. The latency of the CPU-only and GPU-only configuration for our IPsec application has the minimum of 15 μsec and 287 μsec respectively. Though we have different set of modules, it is a promising result as AES encryption and HMAC-SHA1 hashing are heavier computations than radix-tree lookup and string matching because they need to read all payload bytes, encrypt/hash all bytes, and write them back to the packets whereas lookup and string matching only read the packets. Moreover, the average latency of the GPU-only configura-



(a) CPU-only configurations **(b) GPU-only configurations**

Figure 14: NBA's latency distribution. 3 Gbps workload is offered to IPsec and 10 Gbps to others.

tion of NBA's IPv6 router (234 μsec) is roughly same to that of PacketShader (240 μsec) under the same load.

However, the latency distribution of the GPU-only configuration for NBA's IPsec application spreads over a wide range, from hundreds of μsec to a few milliseconds. For example, the average latency is 800 μsec larger than the minimum latency when using 64 B packets. NVIDIA's profiler confirms that the minimum latency observed in our experiment results, 287 μsec, is the minimum possible: the GPU kernel execution takes about 140 μsec (100 μsec for HMAC-SHA1 and 40 μsec for AES-128CTR), and data copies take about 150 to 200 μsec. There is no huge variance in the measured kernel execution latency. Then, we conclude that all additional delays come from NBA and the CUDA runtime.

Our current observations suggest that the primary sources of additional delays are aggressive batching, synchronization overheads in worker-device thread pairs and device-CUDA thread pairs, and queuing delays in both NBA and the CUDA runtime. First, in all GPU-only configurations, the latency is highly sensitive to the batch aggregation size for offloading. The batch aggregation size of 32 batches yields the minimum average latency for IP routers and 64 batches does for IPsec when using 64 B packets. Changing it to other values (twice or half) results in at least 20% of latency increases. Second, as mentioned in § 4.3, the CUDA runtime implicitly spawns an internal GPU handling thread for each runtime context (one per device thread) and uses pthread locks to synchronize. Those locks are potential sources of increased latency. We leave further dissemination of additional delays and optimization as future work.

5 Related Work

Packet processing on commodity hardware: IP routers based on dedicated hardware platforms achieved tens of gigabits per second in mid 90s [40] and today's core routers claim an aggregate speed of hundreds of terabits. Software routers on commodity hardware platforms lag in speed but offer flexible programming environments for quick development and evaluation of new protocols and services. Osiris is one of the early efforts that investigated the performance issues between the network adapters and CPUs [17]. Click offers a modular software architecture for router implementation and has been adopted in many router projects [31].

Egi *et al.* reports on the need of fine-grain control in the forwarding path architecture for task synchronization on software routers [18]. Dobrescu *et al.* reviews pipelining and cloning for parallelism in packet processing and propose an optimization framework that maps data flows to cores. Both use Click [31] as a building block. RouteBricks takes Click to a multi-core PC with 10 Gbps NICs and RB4 has become the first PC-based router to achieve over 10 Gbps speed with four interconnected RouteBricks [15].

The FPGA (Field Programmable Gate Array) technology bridges the gap between customized hardware and commodity technology. NetFPGA is gaining momentum as a developer's platform and has been used by SwitchBlade [9, 34]. Orphal [38] is proposed as a open router platform for proprietary hardware. ServerSwitch [35] uses a custom-designed network card to accelerate certain types of packet processing such as IP lookup with TCAM. Recent developments in Software-Defined Networking (SDN) technologies stress the importance of high-performance commodity-technology-based networking platforms [25, 41].

High-speed packet IO libraries: PacketShader IO Engine (PSIO) [6] is a batch-oriented packet IO API implementation based on Intel's vanilla ixgbe driver. It uses a huge buffer to store multiple packets and thus reduce the kernel-user copy overheads. Its forwarding performance reaches up to 28 Gbps with a single socket of Intel Xeon CPU and four 10 GbE cards. `netmap` is a kernel-space framework for multiple NIC models to offer high-speed packet IO to userspace applications in Linux and FreeBSD [44]. Its baseline performance can saturate the 10 Gbps line rate with minimum-sized packets on a single CPU core running at 900 MHz. Similarly to `netmap`, Intel DPDK [3] offers a framework to develop burst-oriented poll-mode drivers for diverse NIC models. The major difference is that all driver codes are also in the userspace for ease of debugging and development. It can process 80 millions of packets per second using a single socket of latest Intel Xeon CPUs. PF_RING ZC [5] is yet another user-level packet IO library that is developed as a part of packet capture and analysis framework. It reports 14.8 Mpps using a single CPU core of Intel Xeon CPUs.

Virtualization for networking: A handful work have suggested fast packet IO schemes for virtual machines (VMs) sharing physical NICs. NetVM [24] implements shared huge-pages and ring-based packet transfers between VMs

on top of Intel DPDK. ClickOS [7] tailors the kernel for execution of the Click modular router and puts many independent instances of ClickOS VMs to reach line rates. `IX` [11] is a dataplane OS designed to run with Linux control-plane on virtualized environments, with latest hardware supports and optimization. Arrakis [43] uses virtualization to allow direct accesses to the hardware for the raw performance while keeping the same control semantics enforced by the OS. Our work is complimentary to them because NBA is an application-side framework and can be plugged in to other virtualized networking schemes by adding a compatibility IO layer.

New OS abstractions and network stack optimization: Another approach to improve software-based packet processing performance is to tackle the network stacks of operating systems. MegaPipe [22] suggests a light-weight socket optimized for message-dominating workloads. Pesterev *et al.* optimizes Linux's stack to achieve connection locality on multi-core systems [42]. mTCP [28] is an implementation of user-level TCP stack on top of `psio` [6], yielding 33-320% performance improvements. Sandstorm [37] is a clean-slate userspace network stack based on `netmap` [44]. These approaches have advantages in compatibility with existing applications. Our work focuses on packet-level processing (data-plane) instead of flow-level processing or optimization of generic socket applications.

GPUs for specific networking applications: Parallel computing architecture has a long history in supercomputing and has become a key technology in commodity processors as in multi-core CPUs and many-core GPUs. The original target applications for GPUs were in computer graphics and visualization, but are now broadening to many other scientific computations often referred to as GPGPU (General-Purpose computation on GPUs) [1, 39].

PacketShader [21] demonstrates the feasibility of 40 Gbps on a single PC with optimized packet I/O processing and GPU offloading. Gnort [50], MIDeA [51], SSLShader [27], and Kargus [26] all exploit GPU to accelerate network applications, such as SSL (Secure Sockets Layer) and Snort[5]. Our work is a framework to host such specific applications on a unified abstraction.

GPU networking frameworks: GASPP [52] shows an extreme approach to GPU-oriented packet processing. It delivers all packets directly to GPUs via a shared DMA buffer and implements a protocol-oriented modular architecture including transport layers and stateful stream processing. GPUnet [30] is a socket abstraction for GPUs, which allows GPU programs to control RX/TX with remote machines. Snap [47] shares most goals with our work. It uses netmap, adds a set of extensions to Click to integrate GPU elements, and delivers 30.97 Gbps on a single-node quad-core system. NBA is different from them because we treat the CPU as primary processor since it offers low latency and we offload only when GPUs give throughput benefits.

Load balancing and scheduling on heterogeneous processor systems: The problem of finding an optimal schedule for multiple types of tasks on multiple processors is NP-complete, and deadline-based scheduling algorithms cannot be optimal on multiple processors [14, 19, 23, 45]. Qilin [36] targets a set of problems where a small program runs for a long period of time and dynamic compilation at run time is justified for performance over initial overhead of first-run training. It provides an API to be compiled for either or both CPU and GPU and allows runtime adaptation to changing input sizes. Our system targets a very different workload, network traffic, which stresses not only CPU, but also I/O, depending on the traffic composition. Topcuoglu *et al.* [49] study greedy scheduling methods for heterogeneous processor environment. They show that EFT (Earliest Finish Time first) scheduler outperforms in most cases. However, it also requires a performance model and parameters for given tasks before scheduling. StarPU [10] is a generic scheduling framework for systems with heterogeneous processors. It uses heterogeneous earliest finish time (HEFT) scheduling algorithm, which is the best among greedy algorithms, and automatically calibrates the performance model by observing task completion times. Koromilas *et al.* [32] tackles asymmetric scheduling problem of network packet processing workloads running on both integrated GPUs and discrete GPUs. Differently from above work, our framework targets a complex system where the performance of heterogeneous processors have interdependencies to each other and IO as well as computation has critical impacts to the performance.

6 Discussion

Throughput vs. latency: GPU offloading trades off latency with throughput, and is beneficial for specific applications, not all, that require huge parallelism. Our adaptive load balancing currently considers throughput only because latency-oriented optimization would let the system use either the CPU or the GPU only, depending on the application. In this work we have focused on demonstration of the maximum achievable processing capability of our system. A more interesting problem space lies in throughput maximization with a bounded latency, as described in § 7.

Packet reordering: Our current implementation does not guarantee the ordering of packets when using adaptive load balancers, because the CPU and GPU have different processing speeds and NBA transmits the packets as soon as they are processed. However, since NBA drops no packets inside the pipeline except intentionally dropped ones like invalid packets, we expect this will have minimal performance impacts to endpoints if the NICs are configured to use flow control for lossless transfers.

[5] http://www.snort.org

7 Future Work

Migration of existing Click elements: Basing on Click's element abstraction is a huge advantage in that we can migrate existing Click elements. Though, for easier migration we need a wrapper layer for packets because current NBA just exposes DPDK packet buffers directly to elements. Fortunately DPDK's packet buffer API has similar functionality to that of Click's packet objects, and adding thin function wrappers will let only a handful of regular expressions do the job. A more challenging part is to translate `push()` and `pull()` calls to returning the output edge ID. Nonetheless, we expect above tasks would not require major design changes of NBA and can be done in the near future.

Throughput maximization with bounded latency: As we see in § 4.6, offloading to GPUs exhibits high latency in the order of hundreds of μsec. Although there is no standard "acceptable" latency and it depends on what environment NBA is used for, it would be nice if we can limit the maximum latency within a certain threshold because bounded latency makes the system predictable. There may be multiple challenges to tackle, such as designing a new load balancer and/or further optimization of the framework. We are investigating the sources of delays, and guaranteeing bounded latency could be our next step forward.

Extension to other accelerators: NBA's device wrapper (Figure 3) is designed to adapt to multiple different types of accelerators by providing an OpenCL-like interface. Since there is a wide range of devices that support OpenCL, including AMD GPUs and Intel Xeon Phi, we believe that it is viable to expect contributions from the users of NBA. It would be interesting to examine differences of the optimization points and performance characteristics using accelerators with different architectures.

8 Conclusion

We have designed and implemented a software-based packet processing framework for commodity hardware platforms with latest performance optimization techniques. NBA captures underlying architectural details, while providing abstractions of computation batching, GPU offloading and adaptive load balancing to application developers. It delivers up to 80 Gbps performance in IP routing applications and near 30 Gbps in IPsec encryption gateway and a pattern-matching based IDS. We demonstrate that a simple adaptive load balancing scheme can optimize the throughput without manual optimization efforts, even for corner cases where using either CPUs only or GPUs only does not yield the maximum performance. We expect that with 40 Gbps NICs on the market[6] a 100 Gbps software router on a single PC is not far off. We plan to make the source code of the NBA framework publicly available, to motivate development of high-performance software routers.

[6] `http://www.mellanox.com/ethernet/40gbe_index.php`

Acknowledgement

We thank our shepherd Simon Peter and anonymous reviewers for helpful comments, Geoff for framework name suggestions, Sangjin Han for thorough feedback and comments, Shinae Woo and ANLAB members for last-minute reviews, and KyoungSoo Park for general discussion. This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Future Creation and Science (Project No. 2014007580).

References

[1] General Purpose computation on GPUs. `http://www.gpgpu.org`.

[2] NVIDIA CUDA. `http://developer.nvidia.com/cuda`.

[3] Intel® DPDK (Data Plane Development Kit). `https://dpdk.org`.

[4] Perl-compatible Regular Expressions. `http://pcre.org`.

[5] PF_RING ZC (Zero Copy). `http://www.ntop.org/products/pf_ring/pf_ring-zc-zero-copy/`.

[6] PacketShader I/O Engine. `https://github.com/PacketShader/Packet-IO-Engine`.

[7] M. Ahmed, F. Huici, and A. Jahanpanah. Enabling dynamic network processing with ClickOS. In *ACM SIGCOMM*. ACM, 2012.

[8] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18 (6):333–340, 1975.

[9] M. B. Anwer and N. Feamster. Building a fast, virtualized data plane with programmable hardware. In *Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*, VISA '09. ACM, 2009.

[10] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[11] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *USENIX OSDI*, 2014.

[12] G. Chanda. The Market Need for 40 Gigabit Ethernet. `http://www.cisco.com/c/en/us/products/collateral/switches/catalyst-6500-series-switches/white_paper_c11-696667.pdf`, 2012. A white paper from Cisco Systems.

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

[14] E. Coffman and R. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1(3):200–213, 1972.

[15] 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.

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

[17] P. Druschel, L. L. Peterson, and B. S. Davie. *Experiences with a high-speed network adaptor: A software perspective*. ACM,

1994.

[18] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, L. Mathy, and P. Papadimitriou. Forward path architectures for multi-core software routers. In *ACM Co-NEXT PRESTO Workshop*, 2010.

[19] M. Garey and R. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM J. Comput.*, 4(2):187–200, 1975.

[20] P. Gupta, S. Lin, and N. McKeown. Routing lookups in hardware at memory access speeds. In *IEEE INFOCOM*, 1998.

[21] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *ACM SIGCOMM Computer Communication Review*, pages 195–206. ACM, 2010.

[22] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *USENIX OSDI*, 2012.

[23] T. Hu. Parallel sequencing and assembly line problems. *Operations research*, pages 841–848, 1961.

[24] J. Hwang, K. Ramakrishnan, and T. Wood. NetVM: high performance and flexible networking using virtualization on commodity platforms. In *USENIX NSDI*, 2014.

[25] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM*. ACM, 2013.

[26] M. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. Kargus: a highly-scalable software-based intrusion detection system. In *ACM CCS*, 2012.

[27] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: cheap SSL acceleration with commodity processors. In *USENIX NSDI*, 2011.

[28] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a highly scalable user-level TCP stack for multicore systems. *USENIX NSDI*, 2014.

[29] J. Kim, S. Huh, K. Jang, K. Park, and S. Moon. The power of batching in the Click modular router. In *APSYS*. ACM, 2012.

[30] S. Kim, S. Huh, Y. Hu, X. Zhang, A. Wated, E. Witchel, and M. Silberstein. GPUnet: Networking abstractions for GPU programs. In *USENIX OSDI*, 2014.

[31] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The Click modular router. *ACM TOCS*, 18(3):263–297, 2000.

[32] L. Koromilas, G. Vasiliadis, I. Manousakis, and S. Ioannidis. Efficient software packet processing on heterogeneous and asymmetric hardware architectures. In *ANCS*. IEEE Press, ACM/IEEE, 2014.

[33] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: a holistic approach to fast in-memory key-value storage. In *USENIX NSDI*, 2014.

[34] J. W. 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. In *MSE*. IEEE, 2007.

[35] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *USENIX NSDI*, 2011.

[36] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *IEEE/ACM MICRO*, 2009. .

[37] I. Marinos, R. N. Watson, and M. Handley. Network stack specialization for performance. In *ACM HotNets*. ACM, 2013.

[38] J. C. Mogul, P. Yalagandula, J. Tourrilhes, R. McGeer, S. Banerjee, T. Connors, and P. Sharma. Orphal: API design challenges for open router platforms on proprietary hardware. In *ACM SIGCOMM HotNets Workshop*, 2008.

[39] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Eurographics 2005, State of the Art Reports*, Aug. 2005.

[40] C. Partridge, P. Carvey, E. Burgess, I. Castinerya, T. Clarke, L. Graham, M. Hathaway, P. Herman, A. King, S. Kohalmi, T. Ma, J. Mcallen, T. Mendez, W. Milliken, R. Pettyjohn, J. Rokosz, J. Seeger, M. Sollins, S. Storch, B. Tober, G. Troxel, D. Waitzman, and S. Winterble. A 50-Gb/s IP router. *IEEE/ACM Transactions on Networking*, June 1998.

[41] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, et al. Ananta: cloud scale load balancing. In *ACM SIGCOMM*. ACM, 2013.

[42] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *EuroSys*. ACM, 2012.

[43] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *USENIX OSDI*, 2014.

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

[45] J. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. *Computer*, 28(6):16–25, 1995.

[46] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[47] W. Sun and R. Ricci. Fast and flexible: parallel packet processing with GPUs and click. In *ANCS*. ACM/IEEE, 2013.

[48] K. Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 1968.

[49] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260 –274, mar 2002. ISSN 1045-9219. .

[50] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. Markatos, and S. Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *RAID*, 2008.

[51] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. MIDeA: A multi-parallel intrusion detection architecture. In *ACM CCS*. ACM, 2011. ISBN 978-1-4503-0948-6. . URL `http://doi.acm.org/10.1145/2046707.2046741`.

[52] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis. GASPP: a GPU-accelerated stateful packet processing framework. In *USENIX ATC*. USENIX Association, 2014.

[53] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing lookups. In *ACM SIGCOMM*, 1997.

[54] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, high performance ethernet forwarding with CUCKOOSWITCH. In *ACM CoNEXT*, 2013.