Design of a Scalable Network Programming Framework

Ben Wun
Washington University
Computer Science and Engineering
St. Louis, MO 63130-4899
+1-314-935-4203
bw6@cse.wustl.edu

Patrick Crowley
Washington University
Computer Science and Engineering
St. Louis, MO 63130-4899
+1-314-935-9186
pcrowley@wustl.edu

Arun Raghunath
Intel Research and Development
Hillsboro, OR 97124
+1-503-264-4892
arun.raghunath@intel.com

ABSTRACT

Nearly all programmable commercial hardware solutions offered for high-speed networking systems are capable of meeting the performance and flexibility requirements of equipment vendors. However, the primary obstacle to adoption lies with the software architectures and programming environments supported by these systems. Shortcomings include use of unfamiliar languages and libraries, portability and backwards compatibility, vendor lock-in, design and development learning curve, availability of competent developers, and a small existing base of software. Another key shortcoming of previous architectures is that either they are not multi-core oriented or they expose all the hardware details, making it very hard for programmers to deal with. In this paper, we present a practical software architecture for high-speed embedded systems that is portable, easy to learn and use, multi-core oriented, and efficient.

1. INTRODUCTION

Programmable network devices, often implemented using network processors (NPs), offer programmers the performance they need to implement high-speed networking applications. This performance comes at a cost; most NPs expose a great deal of architectural details to the programmer. Furthermore, these details vary greatly among different processors offered by different vendors. The creation of a standard, portable, easy-to-use framework would ease the process of developing software on these systems. It is our belief that more widespread adoption of these platforms is hindered not by performance considerations, but by the difficulty of programming them. Many equipment vendors would be happy to trade reasonable amounts of performance or efficiency for advantages such as improved programmer productivity, a larger base of software and developers, and an expectation of better application portability to future systems.

In our view, no current programming environment provides the ideal set of characteristics for a networking device. These include good performance, portability, reuse of familiar languages, ease of use, and backwards compatibility. Because the needs of the network programming community are not being met by existing solutions, we see an opportunity to provide a programming environment that will be beneficial to network operators, system vendors, semiconductor vendors, and software developers alike.

We present a design for a framework that can fill this need, called the Network Runtime Environment (NRTE). We also examine the performance of a prototype implementation of the NRTE for Linux-based machines. In Section 2, we examine existing network programming interfaces and the need for a new framework. In Section 3, we present the design of the NRTE along with micro-benchmarks demonstrating the benefits of the NRTE features. In Section 4, we examine the performance of several example applications based on the NRTE. We discuss future work in Section 5 and conclude in Section 6.

2. MOTIVATION AND RELATED WORK

This section examines the need for an efficient and portable network application programming framework. We will discuss the interested parties, the shortfalls of existing frameworks, and the features necessary for a successful framework.

2.1 Interested Parties

A network programming framework that is easy to use and portable would benefit many parties, including semiconductor vendors, equipment vendors, network operators, and third-party software developers.

Semiconductor vendors would benefit from a good framework that makes their chips easier to program. This would attract more programmers and make it more cost-effective for equipment vendors to adopt their products. This would, in turn, lead to an increase in the number of chips that semiconductor vendors can sell.

Equipment vendors could produce their platforms using fewer chips by implementing functionality in software, and their investments in software would carry forward to new generations of chips. By implementing most functionality in software, they could more easily maintain their existing base of applications and increase product differentiation by adding features not available out of the box.

Another party that would benefit is network operators. They could avoid vendor lock-in, which tends to diminish competition and available choices. They could also deploy services and features in software without waiting for equipment vendors to implement them, which is often a slow process not under their

	Performance	Portable	Familiar Language	Ease of Use	Backwards Compatible	Multi-core-oriented
Click	X			X	X	
Shangri-la	X		X	X		X
NetVM		X				X
sockets		X	X	X	X	
Autoparitioned IXP C	X		X			X
Juniper PDSP	?		?	?	?	?

Table 1: Framework comparison

control. Furthermore, the ability for network operators to add their own features increases differentiation. If they must wait for equipment vendors to include a new feature in their products, their competitive advantage is eroded, because anyone can then deploy that service.

Finally, third-party software developers would be able to sell more software if there were a single framework that was compatible across products offered by different vendors. [10].

2.2 Existing Frameworks

There exist many APIs and programming frameworks that solve some aspect of the problem we are examining. Frameworks for network applications that run on general-purpose processors and standard OSes include Sockets and Click [3].

Sockets are the standard API for network programming in UNIX-based systems. It is popular and widely used, but maintaining the interface has become a bottleneck in the performance of network processing applications. Furthermore, Sockets are designed to run on a standard PC, whereas different programming paradigms are used in most network devices.

The Click framework provides a library of predefined elements for common networking tasks that can be composed into a complete application using the Click language. Additional functionality can be added by writing new elements in C++ that conform to the Click framework. One disadvantage is that it requires a programmer to learn the Click language to write applications.

Another disadvantage of Click is that the framework for writing new elements is written in C++, which is not usually available for embedded platforms such as network processors. While Click configurations are potentially portable if a Click implementation that includes the elements used in the configuration is already available for the target platform, the implementation of new elements is not. Implementing new protocols (for example, [22]) not supported by existing Click elements is difficult to do in a portable fashion. The NP-Click [4] project created a Click implementation for the IXP network processors, but the underlying element implementations were rewritten in IXP-C, and the interface for writing new elements is incompatible with the original Click.

There are various solutions for easing the programming of NPs that abstract away some of the difficulties of programming in a low-level environment. These include Shangri-la [5] and the Intel auto-partitioning compiler [6]. These are C-based solutions, but they both run only on Intel's IXP network processor. Shangri-la's Baker programming language is platform-independent, but it is not truly portable because no implementations exist for other platforms.

Another proposed solution is NetVM [1], a virtual machine for network processing. NetVM attempts to define a virtual machine with its own virtual instruction set. An interpreter or compiler can take this generated byte code and run it on a specific platform. The authors define an architecture for this virtual machine, but not a programming model. Moreover, the prototype performs rather poorly. It took 2236 clock cycles to perform an IPv4 filtering task that took a Berkley Packet Filter implementation 124 clocks to perform.

Recently, Juniper began offering third parties the ability to program their routers through their Partner Solution Development Platform (PSDP) [11], which provides a framework on top of the JUNOS operating system running on their routers. This solution apparently provides a uniform framework for all Juniper routers, but there is not much information publicly available. However, it demonstrates that companies are beginning to see the advantages of opening up their router platforms to third-parties.

Finally, projects such as XORP [20] pursue a complementary goal. XORP is an interface for control-plane processing, which runs above the data plane. XORP can work using different data-plane implementations, including Click. It could also potentially implement the data plane using the interface we will introduce in this paper.

Table 1 summarizes the frameworks discussed above, and highlights which characteristics of a good networking framework (discussed in the next section) they exhibit. As can be seen, no framework fulfills all these requirements.

Processor (NP=Network Processor GPP = General Purpose Processor)	Clock (MHz)	CPU Power (W)	Chipset Power (W)	Packet I/O (Gb/s)	Mem I/O (Gb/s)	Processor Cores	Core Issue Width	Peak BIPS	Peak BIPS/W
Cisco SPP (NP)	250	35	0	192	175	188	1	47	1.34
Intel IXP2855 (GPP)	1500	27	0	25	121.6	16	1	24	0.89
Cavium Octeon CN5860 (NP)	1000	40	0	25	102.4	16	2	32	0.80
Raza XLR 732 (NP)	1000	32	0	25	230.4	8	1	8	0.25
Cavium Octeon CN3860 (NP)	600	30	0	25	102.4	16	2	19.2	0.64
Intel Quad-Core Xeon 5300, Intel 5000P chipset (GPP)	2330	80	30	0	85.6	4	4	37.28	0.34
AMD Dual-Core Opteron 1218 HE (GPP)	2600	65	0	192	85.6	2	3	15.6	0.24
Niagara 2 (NP)	1400	84	0	40	307.2	8	2	22.4	0.27

Table 2: Processor characteristics

2.3 Requirements of Framework

2.3.1 *Assumptions*

The goal of this paper is to present a framework for programming network devices. In this section, we will present the features we believe must be included in any successful framework.

We will first lay out the assumptions that will affect our design. First, we assume that current and future programmable network devices have the capacity to perform their tasks at acceptable performance levels. Table 2 details the characteristics of several current network processors (data from [8][7][12][15][14]) compared with several general-purpose chip multiprocessors (data from [17][16][13]). We can see that the NPs provide significantly more memory I/O bandwidth. There is no significant difference in peak processing power between the NPs and the general-purpose processors (expressed in billions of instructions per second or BIPs). But the NPs, which have simpler cores and more programmer control over resource utilization, are more likely to achieve their peak BIPs. Due to increased I/O capacity and greater resource utilization, these NPs are capable of achieving the line rate processing that general-purpose processors cannot.

Second, we assume that in the future, new generations of NPs will increase performance by the addition of more concurrent threads and new hardware accelerators, not by any appreciable increase in clock speed. This is again borne out: Table 2 shows that the network processors all contain significantly more cores than the general-purpose processors, while at the same time maintaining lower clock rates. This indicates a requirement for a networking framework to be able to leverage more cores and hardware threads.

2.3.2 *Performance*

Network devices are designed with enough raw power to perform the tasks they are designed for, but a poor software layer can significantly reduce performance. A major goal of a good framework should be that it can be easily implemented on a number of systems without unduly strangling performance. For example, a major shortcoming of the NP-Click [4] project is that small-packet performance is greatly reduced compared with that on an application written in the IXP's native C-derivative.

2.3.3 *Portability*

There are myriad platforms on which a networking application may run, and even more are likely to be developed in the future. Allowing programmers to write code that can easily be moved to another platform helps to future-proof it as well as prevent vendor lock-in. However, making a framework generic enough to be portable may prevent the use of platform specific accelerators. For example, NPs like Cavium's Octeon [7] contain fixed function accelerators for cryptographic and hash functions. A standard software interface for these units will have to be included in the framework, with functionality provided in software on platforms that do not contain these accelerators. This is the strategy used by the designers of OpenGL to solve the problem in the graphics domain.

Furthermore, portability among platforms offered by different vendors is desirable. Being locked in to a single vendor is undesirable for many reasons; for example, if that vendor decides to discontinue a particular line of products, the investment made in writing software for that architecture is suddenly voided. Vendor independence reduces the risk of investing in network

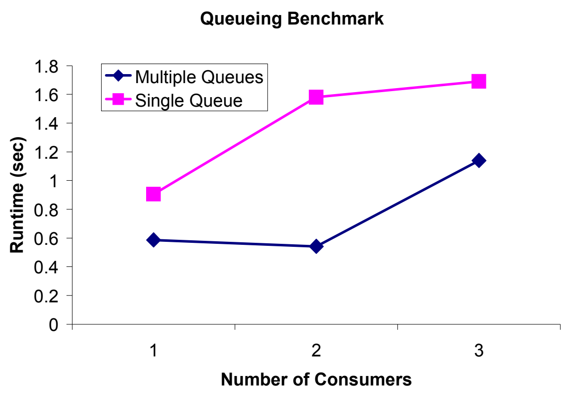


Figure 1: Queuing Benchmark

software development and encourages broader adoption of these platforms.

2.3.4 *Choice of programming language*

Another aspect that must be considered is the language used in the framework. Writing a domain-specific language has its advantages because such a language can be tailored to the needs of the networking domain. Several languages and compilers have been designed for the Intel IXP network processor [8], and others, such as Click [3], for general-purpose systems. However, each of these languages has a learning curve that limits the pool of developers who are capable of programming these devices.

The other possibility is to have the framework provide a runtime and/or libraries that are compatible with the use of existing languages. Most NPs have some sort of C derivative that they can be programmed in. C is a common language understood by many programmers, and if the framework is based on C (or some other widely known language), this makes the platform more easily programmable by a large pool of programmers.

2.3.5 *Ease of use*

One of the reasons that network devices are difficult to program is that they often expose low-level architectural details to the programmer. Abstracting away these low-level details, which are different for every processor, will ease the programmer's burden. Furthermore, libraries that perform common tasks will likewise increase programmer productivity.

However, ease of use cannot be allowed to unduly hinder performance. An example of this is the sockets framework which is popular and widely adopted. However, the sockets framework is often a bottleneck. The synchronous nature of the framework makes it hard for programmers to express the inherent parallelism of many applications.

2.3.6 *Backwards compatibility*

It is not desirable for significant changes to the programming model in future processors to nullify the investment in software of the last generation. In networking equipment, the majority of a system's intellectual property, competitive advantage, and overall complexity are found in the software rather than the hardware. If the software model changes drastically, it erodes this investment in software. This has already happened with revisions to the IXP's programming model. An effective framework that is portable across architectures will help ease this problem.

Because we expect future generations of networking devices to get a performance boost from additional threading and fixed-function accelerators, the ability to take advantage of these new features automatically is paramount. How this can be done is an open question. Some attempts have been made to find ways to automatically map program components to hardware threads [6][5]. None of these solutions is perfect. First, there is a tradeoff between static mapping at compile time and automatic remapping at runtime. The former obviates the need to have a runtime system that can remap program components, which can potentially require significant overhead. The latter would be better at adapting to changing workloads. As demonstrated in [9] the ability to adapt to different workloads can significantly improve performance in the face of changing workloads.

Future proofing is difficult; if we make the assumption that future performance comes from greater parallelism, the proper way to exploit that parallelism automatically is not clear. One possibility is to organize programs in a pool of worker threads, and to scale up the number of program threads as the available number of hardware threads increases, but not all applications are amenable to this type of parallelization. Programs that use pipelined parallelism and distribute their stages among multiple hardware threads can only scale to a point; once the number of hardware threads exceeds the number of explicit stages in the program, these applications will no longer automatically benefit without programmer intervention. At the very least, programmers must either change their code to create more duplicate threads, or write extra code to allow their programs to adapt automatically, but this is cumbersome and done on a case-by-case basis. A good framework should take care of this automatically.

3. NRTE

The Network Runtime Environment (NRTE) is our implementation of a multi-core oriented, network programming environment. We have designed it to meet the criteria for a successful framework that we outlined above. It is implemented as a C library, which presents programmers with a familiar programming language that is portable across many platforms. It is explicitly designed to allow programmers to expose as much parallelism as possible in their programs while leaving the details of mapping to the underlying hardware threads up to the runtime. Finally, as we will show in Section 4, performance is comparable to existing solutions in this domain.

The NRTE requires the user to handle parallelism—it is the programmer's responsibility to break the application into stages and to make these stages thread-safe. The application is organized as a pipeline with stages communicating with each other over queues. The runtime is responsible for mapping these stages onto the underlying hardware. Additional parallel computing resources are taken advantage of by duplicating stages and splitting the incoming packet flows among these duplicates. The hardware mapping decision is left until runtime, when the user can specify the hardware mapping using a control program.

The NRTE provides two types of stages: explicit and implicit. The explicit stages are threads under explicit user control; they launch and run to completion much like a thread created using the pthreads library. Implicit stages are registered as callback functions on an associated queue. The registered function is called by the runtime to process elements on that queue. The instantiation of implicit stages is left to the runtime; if the user indicates that the implicit stage is safe to duplicate (i.e., is thread-safe), the runtime can create duplicates of that stage to run on multiple hardware threads. Packets are distributed among the duplicates using flow-pinning. The user specifies the flow definition via a classification function. The runtime uses this function to send packets of the same flow to the same stage, so they can be processed in order and take advantage of cache locality for flow-specific data structures. Parallelism is achieved by processing different flows on different threads. The use of implicit stages in this manner allows us to scale the application to take advantage of increasing numbers of cores on future processors.

The NRTE strategy for dealing with multi-threading differs from Click's. Click creates a task list for its configuration and load-balances the tasks across cores. This does not take into account caching effects, and it has a fixed amount of parallelism. If there are more hardware threads than tasks, Click cannot make use of them. By repeatedly duplicating the bottleneck stage(s), the NRTE actually creates parallelism, which can potentially scale to as many threads as are provided by the hardware.

We confirmed the usefulness of flow-pinning by creating an application that mimics the access pattern of stateful packet-processing applications. This application receives incoming elements and accesses and modifies state associated with that element's flow. We measured the time it took the application to process a fixed number of elements; we found that when flow-pinning was used to distribute the incoming data, the application ran in 12.04 seconds. When the incoming data was distributed without regard to flow, it took 16.19 seconds. This is due mostly to cache-thrashing. Our micro-benchmark does not include the cost of locking, which would likely be needed to prevent data corruption in a real application. Including this would further improve the flow-pinning results because it eliminates the lock synchronization overheads incurred without flow-pinning when multiple threads attempt to access the same flow state.

When a stage is duplicated, the underlying queues used to connect that stage to the rest of the pipeline are also duplicated. This is done to prevent synchronization overheads that would result if there were only one underlying queue with multiple replicas trying to read from it. When an implicit stage is duplicated by the runtime, there will be more than one consumer on that stage's queue. A simple way to maintain integrity of the underlying queue would be to use mutual exclusion locks to allow atomic access to a single producer or consumer at a time, but we observed that this quickly became the program's bottleneck. Instead, by implementing the logical queue abstraction as a set of point-to-point queues with a single producer and a single consumer each, we remove this necessity. The underlying physical queues do not require locks. Figure 1 shows a micro-benchmark we constructed comparing the use of the NRTE's queuing strategy of single consumer/producer circular buffers with an implementation where a single queue (still a circular buffer) is shared among multiple consumers, requiring synchronization with locks. The program has a single producer enqueueing items onto a queue and we measured the time it took for the consumers to dequeue all of them. The producer was producing while the consumer was simultaneously consuming. We varied the number of consumers from one to three, mapping a single producer or consumer to each core in our test system until we ran out of cores. The data show that the single-queue implementation, which requires lock synchronization for each enqueue or dequeue, performs far worse than the multiple-queues implementation. In addition, as we vary the number of consumers, the single-queue implementation scales poorly. The multiple-queues implementation scales well; no change in performance is observed between the one- and two-consumer cases. When moving to three consumers, performance degrades, but not for queuing reasons; the third consumer is placed on a core that shares an L2 cache with the producer, resulting in cache-capacity problems.

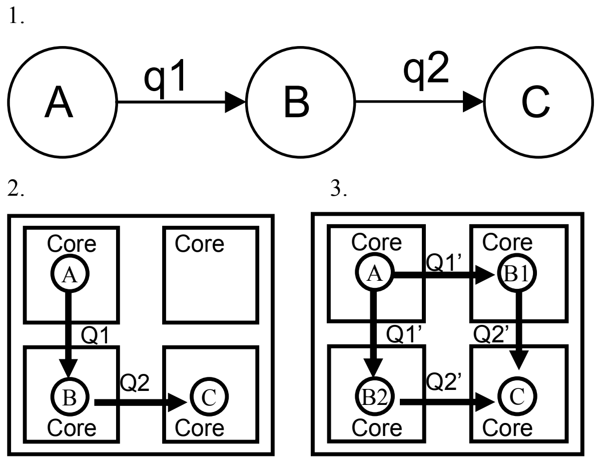
The NRTE also includes a packet abstraction and libraries to deal with common packet-processing tasks. Packet-handling abstractions allow the user to write platform-independent code, so that running NRTE-derived programs that must run in different environments does not require extensive rewrites. For example, in different situations, the same application may be run as a user space program that manipulates packets using sockets; as a Linux kernel module that directly manipulates packets in skbuffs; or on an IXP network processor, which uses neither. Libraries that provide common functions such as check-summing or efficient algorithms for longest prefix match will allow programmers to concentrate on what is unique to their programs rather than reinventing the wheel. Finally, putting efficient implementations for common data structures into the framework provides both the convenience and the ability to take advantage of underlying hardware support without programmer intervention. An example of this is inter-core communication using queues. Hardware support exists for this on specialized platforms such as the IXP. On x86 platforms this is not present, but efficient queue implementations, described above, are used.

We will illustrate the workings of the NRTE with an example. Our example begins with the following code snippet:

```
main()
{
    a = A();
    b = B(a);
    C(b);
}
```

Where A(), B(), and C() are functions that perform packet-processing operations. This can be turned into a pipeline, with dataflow illustrated in Figure 2 using the NRTE with the following code:

```
nrte_queue_id_t q1, q2;

void A(void *ignored)
{   //Do A's work, then send on
    uint64_t A_elem;
    nrte_enqueue(q1, A_elem);
}

void B(uint64_t stage_id, unsigned int flow_id,
uint64_t A_elem)
{  //Do B’s work, then send on
   uint64_t B_elem;
   nrte_enqueue(q2, B_elem);
}

void C(uint64_t stage_id, unsigned int flow_id,
uint64_t B_elem)
{  //Do C’s work
}

unsigned int flow_classifier(uint64_t elem)
{   //dummy function. Simply returns input
    return (unsigned int)elem;
}

main()
{
    //Create the flow graph in the NRTE
    q2=nrte_register_queue(C,flow_classifier);
    q1=nrte_register_queue(B,flow_classifier)
    nrte_register_explicit_input_function(A);

    nrte_start();
}
```



1. **Programmer-specified application structure.**
2. **Pipeline can be mapped to 3 cores.**
3. **Same pipeline can be mapped to 4 cores without modifying the program. Queues into and out of thread-safe replicated stage are automatically duplicated so each has a single producer and consumer, thus avoiding locks.**

Figure 2: NRTE Dataflow

This code creates an explicit stage to run the first calculation, A(), and two implicit queues to calculate B() and C(). When A() finishes its calculation, the result is enqueued to the second stage, which performs calculation B(), which in turn passes its output to stage C(). A dummy flow classifier function is defined, which will be called if the implicit stages are duplicated to help the runtime decide which copy to pass it to. The main function is responsible for registering the stages with the runtime and then kicking things off by calling rte_start(). Once this happens, control passes to the runtime, which will start the explicit stage, A(), and run the implicit stages B() and C() when items appear on their respective queues. The three stages would most likely be run on three different hardware cores or threads.

Figure 2 shows the mapping of the logical pipeline to hardware. The first configuration maps a single copy of each stage to a separate core. The next panel illustrates what happens if the runtime decides to replicate the second stage, B. The stage is copied and the second copy mapped to another hardware thread. The queues associated with it are also duplicated; stage A will see one logical queue, (q1) but the underlying implementation is two single-producer, single-consumer queues (the q1’s). The runtime takes care of deciding on which queue to put an element enqueued from A. The same is true at the other end; each copy of B has its own queue to C, although to the single thread running C, it looks like a single logical endpoint. The use of separate point-to-point queues in the underlying implementation allows these queues to be implemented efficiently and without locking—which, as we demonstrated earlier, can quickly become a bottleneck if there are multiple producers and/or consumers.

Note that the duplication and mapping of the stages is done at runtime by a control program and is not expressed in the application itself. This makes it easy for the programmer to make their code independent of the number of cores on the system; mapping to hardware is done at runtime. Furthermore, the programmer can experiment with different mappings to find the optimal mapping for a given platform or workload.

4. PROTOTYPE PERFORMANCE

We have implemented a prototype of the NRTE that runs on x86 Linux systems. This prototype implements the NRTE threading model and packet-handling abstractions, and includes a control program to allow the user to manually map program stages to hardware threads at runtime. Furthermore, the standard libraries include implementations of elements and algorithms such as queues, packet buffers, checksum algorithms, and longest prefix match. This section shows that the NRTE prototype demonstrates the qualities of a good framework that we outlined above without sacrificing performance.

4.1 Test Setup

Our test system uses dual, dual-core Xeon processors (a total of 4 cores), and a 4 port e1000 network card, running Linux with a 2.6.20.1 kernel. Another machine using the Linux kernel’s pktgen module is used as a traffic generator. Forwarding rates were measured at the receiving end, using a receiver that is capable of receiving packets faster than the test system is able to forward them.

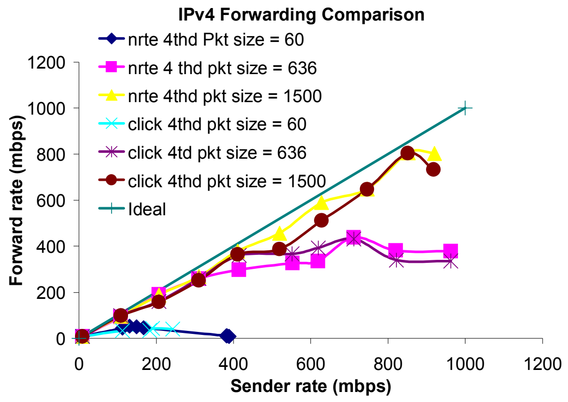


Figure 3: IPv4 Forwarding Comparison

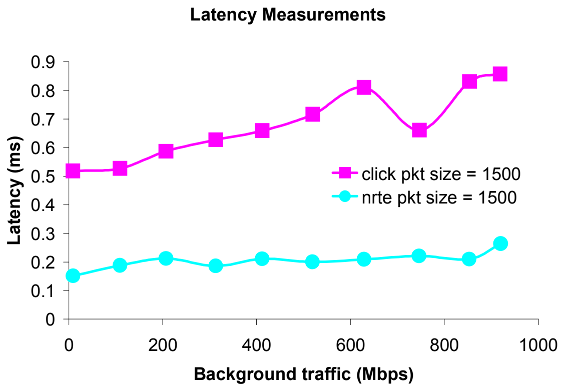


Figure 4: Latency

For our tests, we measured implementations of IPv4 forwarding and NAT, written using the NRTE, against Click configurations for the same application running in user space. Click performs better running as a kernel module, but the kernel module version of the NRTE is still in development. However, the user-space test is still meaningful as a comparison, and is the mode that must be used in platforms such as Planetlab [21], that do not allow programmers to run code in the kernel for security and isolation reasons. Environments like Planetlab are an important target, as they can be used to prototype new protocols, and if a portable framework like the NRTE is used in the prototype, moving to more high-performance platforms will be easier.

4.2 IPv4 Forwarder

Our implementation of the IPv4 forwarder is functionally equivalent to the Click router configuration we used. They use the same algorithm for longest prefix match (DIR 24-8 BASIC) [18], which performs efficient lookups in most 2 memory accesses.

The NRTE forwarder uses two pipeline stages: an RX stage that receives and classifies packets, and a forwarder stage that does most of the forwarding work (checksum computation, route lookup). RX is an explicit stage; there is only one copy, which is instantiated by the user. The forwarder stage is an implicit stage. At runtime, we duplicated the forwarding stage and ran it on three different cores, with packets split by flow (defined in this case by the destination address) among the copies. While no flow-specific data structures are necessary in this application, flow-pinning results in in-order packet processing for each flow, which has implications for protocols such as TCP. The duplication and mapping of the forwarder stage is done at runtime by a control program and is not expressed in the application itself. This makes it easy for the programmer to make their code independent of the number of cores on the system because the mapping does not have to be present in the code. Furthermore, the programmer can experiment with different mappings to find the optimal mapping for a given platform or workload. This might even be done automatically by the runtime in future versions of the NRTE.

We measured forwarding rates for each application for a variety of packet sizes and sender rates. The results are shown in Figure 3. This data clearly show that over a wide range of packet sizes and sender rates, the NRTE IPv4 forwarder performs as well as or better than the Click configuration.

4.3 Latency

In our next experiment, we examined the latency of packets traversing our IPv4 router implementations. The experiment was set up the same as that in Section 4.2, but this time, a ping was sent from one machine connected to the router system on a different port to another machine also connected to the router. This was done in the presence of background traffic generated in the same manner as in the previous IPv4 forwarder experiment. The latency for a reply was measured at the ping sender. Figure 4 shows the results of this experiment for the Click router and the NRTE router. The NRTE-based router produces much lower round-trip ping times than the Click router.

4.4 Scaling

We also tested the scaling of the IPv4 forwarder as we increased the number of threads. Because we want to stress the NRTE code and do not want to be bottlenecked by the sockets layer, we modified the code to process a stream of pre-allocated dummy packets. We measured the time it took for the program to process a fixed number of packets. We found that when increasing from two threads (one Rx thread, one forwarding thread) to three (adding a second forwarding thread), runtime decreased from 18 seconds to 13 seconds, a ratio of 0.77, compared with an ideal halving of processing time. This is due to the bottleneck of the Rx thread, which cannot be duplicated. After adding a fourth thread (third forwarding thread), performance deteriorated to 45 seconds due to having the additional thread running on a core that shared an L2 cache with the Rx thread.

We added a busy loop to increase the processing time of the forwarding stage so that the Rx stage would no longer be the bottleneck. This caused the ratio of the two-thread to three-thread mappings to be 0.49, which is very close to the ideal of 0.5.

4.5 NAT

Our second test application is network address translation (NAT). We tested a Click configuration using the IPRewriter class and an NRTE-based implementation. The NRTE implementation used a single receive stage and three NAT stages that did most of the work. We tested the rewriting of IP destination addresses and

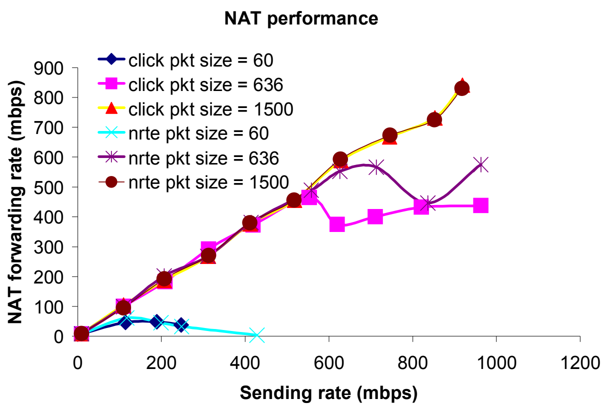


Figure 5: NAT comparison

UDP destination ports. Figure 5 shows the results. We can see that, once again, the NRTE and Click versions of the same application perform comparably.

4.6 Snort

We ported Snort[19], a popular open-source intrusion-detection program, to work with the NRTE. We divided Snort into four functional blocks—a packet receive and classification stage run as an explicit stage, and three implicit stages: a preprocessor stage that runs Snort preprocessors such as TCP stream reassembly and portscan detection, a detect stage that performs regular expression-matching, and a logging stage that logs the results. The three implicit stages are written to be thread-safe so they can be replicated. Packets flow from stage to stage in the order they have been listed, with packets in different stages running in parallel on different cores.

The pipelined nature of this application allows us to demonstrate one of the strengths of the NRTE. By leaving mapping decisions to runtime, we were able to test different pipeline configurations to find the best one for each workload. For some workloads, replicating the preprocessor stage improved performance; for others, replicating the detect stage was the better option. Having a control program that allows the programmer to decide the program's mapping to hardware at runtime without rewriting the program, is thus an extremely useful tool.

The Snort experiments were executed on an 8 core system (dual quad core CPUs), with packet traces read off a disk to avoid having the network become a bottleneck because we were interested in the performance of the Snort program, not the network stack. Furthermore, dropping packets can cause major changes in Snort's behavior and reading the trace from a file allows Snort to throttle its own input rate. We measured the total time Snort requires to process a trace and used that as the metric for comparison.

Our traces were collected from connections between university networks and the Internet core (NLANR). We believe these to be a good sampling of real-world workloads likely to be encountered by Snort. Because these traces include only packet headers, synthetic packet bodies were inserted for testing.

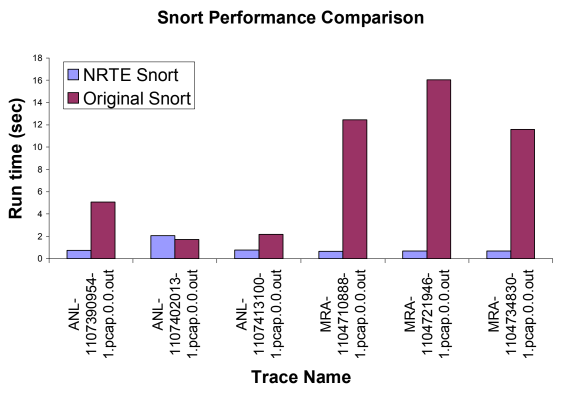


Figure 6: Snort comparison

The results are in Figure 6. As can be seen from this graph, the multi-thread enabled NRTE-Snort significantly outperforms the single-threaded original Snort in processing all workloads. While the speedup of the ANL traces can be attributed to the parallelism that we were able to extract via pipelining, the big speedups in processing the MRA traces are due to the large number of flows (~25000) in these traces. We were able to leverage this parallelism in the data by replicating the TCP stream reassembly stage (the bottleneck stage) and pinning subsets of the flows to each replica. In this fashion, the NRTE allows more hardware resources to be effectively applied to the application bottleneck.

5. FUTURE WORK

We have shown that our prototype of the NRTE for user space x86 can perform as well as the most popular existing solution (Click). However, many of our goals remain unmet. One of the main rationales for the NRTE is portability, and while the system can be ported to almost any embedded system with standard C-compiler support, our next major milestone is to port the NRTE to other platforms, such as a network processor, that have near-standard C-compiler support. We have shown that the use of a control program for manual runtime mapping of stages to cores is extremely useful; a further optimization along this line would be to include an option for the runtime to find the optimal mapping of stages to hardware automatically. This is especially true if the optimal configuration changes over time as a result of a changing workload and the runtime can remap the program dynamically to compensate. The usefulness of such an approach is demonstrated in [9].

6. CONCLUSION

The creation of an open, portable, easy-to-use framework for programming network processing systems is important for getting the most out of these systems. It is advantageous to programmers, who will be able to more easily write programs for these platforms; and will be a boon to device manufacturers, who will be able to sell more systems. Network operators and equipment vendors will also benefit.

We have presented our solution, the NRTE, a portable, C-based, multi-core oriented framework and have demonstrated that it performs comparably to Click. The NRTE provides great flexibility to the user for specifying parallelism and provides a method for scaling up the exposed parallelism by duplicating stages. Furthermore, it provides flow-pinning and transparent queue management to make interstage communication simple and efficient. All these mechanisms provide a high-performance, portable, easy-to-program framework for network programming in multi-core environments.

7. ACKNOWLEDGMENTS

This work was supported by NSF grant CCF-0430012.

8. REFERENCES

[1] L. Degioanni, M. Baldi, D. Buffa, F. Risso, F. Stirano, G. Varenni. "Network Virtual Machine (NetVM): A New Architecture for Efficient and Portable Packet Processing Applications." In *Proceedings of the 8th International Conference on Telecommunications, 2005 (ConTEL 2005)."* June, 2005.

[2] John Giacomoni, John K. Bennett, Antonio Carzaniga, Douglas C. Sicker, Manish Vachharajani, Alexander L. Wolf. "Frame Shared Memory: Line-Rate Networking on Commodity Hardware In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, Orlando, FL, 2007.

[3] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. "The Click Modular Router," ACM Transactions on Computer Systems **18**(3), August 2000, pages 263-297.

[4] N. Shah, W. Plishker, and K. Keutzer. *NP-Click: A programming model for the Intel IXP*1200. In Proceedings of 9th International Symposium on High Performance Computer Architectures (HPCA), 2nd Workshop on Network Processors, February 2003.

[5] Michael K. Chen, Xiao Feng Li, Ruiqi Lian, Jason H. Lin, Lixia Liu, Tao Liu, Roy Ju. "Shangri-La: achieving high performance from compiled network applications while enabling ease of programming," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation,* Chicago, IL, 2005.

[6] Jinquan Dai, Bo Huang, Long Li, Luddy Harrison. "Automatically partitioning packet processing applications for pipelined architectures," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation,* 2005.

[7] Cavium Networks. OCTEON CN38XX/CN36XX Multi-Core MIPS64 Based SoC Processors, http://www.cavium.com/OCTEON_CN38XX_CN36XX.htm l

[8] Adiletta, M., et al. "The Next Generation of Intel IXP Network Processors," Intel Technology Journal, vol. 6, no 3, pp. 6-18, Aug 2002.

[9] Arun Raghunath, Aaron Kunze, Erik Johnson, Vinod Balakrishnan. "Framework for supporting multi-service edge packet processing on network processors," In *Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems table of contents*, Princeton, NJ, 2005.

[10] Patrick Crowley, Derek McAuley, Thomas Woo, Jon Turner, Chuck Kalmanek. "Open Router Platforms: Is it time to move to an open routing infrastructure?" ANCS 2007 panel session. http://www.cse.wustl.edu/ANCS/advance_program.html#Pan el

[11] Juniper Networks :: Open IP Solution Development Program. http://www.juniper.net/partners/osdp.html

[12] Cavium Networks, OCTEON CN38XX/CN36XX Multi-Core MIPS64 Based SoC Processors, http://www.cavium.com/OCTEON_CN38XX_CN36XX.htm l

[13] AMD, Inc., AMD Opteron Processor Product Data Sheet, http://www.amd.com/us-en/Processors/TechnicalResources/0,,30_182_739_9003,00. html

[14] RMI: XLR Family of Thread Processors. http://www.razamicro.com/products/xlr.htm

[15] Eatherton, Will. "The Push of Network Processing to the Top of the Pyramid," ANCS 2005 keynote address, Oct 27, 2005. http://www.cesr.ncsu.edu/ancs/slides/eathertonKeynote.pdf

[16] Intel Corp., Intel Quad-Core Xeon 5300 and Intel 5000P chipset data sheets, http://www.intel.com/design/intarch/quadcorexeon/5300, http://www.intel.com/products/chipsets/5000p.

[17] Umesh Gajanan et. al. "An 8-core, 64-thread, 64-bit, power efficient SPARC SoC." http://www.opensparc.net/pubs/preszo/07/n2isscc.pdf

[18] Pankaj Gupta, Steven Lin, Nick McKeown. "Routing Lookups in Hardware at Memory Access Speeds." In Proc. IEEE Infocom 1998, Vol. 3, pp. 1240-1247.

[19] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA Systems Administration Conference,* Nov. 1999. (http://www.snort.org).

[20] Mark Handley, Eddie Kohler, Atanu Ghosh, Orion Hodson, Pavlin Radoslavov. "Designing Extensible IP Router Software." In Proceedings of the 2nd USENIX Symposium on Networked Systems (NSDI) 2005.

[21] Chun, B., D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. "PlanetLab: An Overlay Testbed for Broad-Coverage Services," *ACM Computer Communications Review,* vol. 33, no 3, 7/03.

[22] Stoica, I., D. Adkins, S. Zhuang, S. Shenker, S. Surana, "Internet Indirection Infrastructure," Proc. Of ACM SIGCOMM, 8/02.