Implementation of a Simplified Network Processor

Qiang Wu, Danai Chasaki and Tilman Wolf
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA, USA
{qwu,dchasaki,wolf}@ecs.umass.edu

***Abstract*—Programmable packet processors have replaced traditional fixed-function custom logic in the data path of routers. Programmability of these systems allows the introduction of new packet processing functions, which is essential for today's Internet as well as for next-generation network architectures. Software development for many existing implementations of these network processors requires a deep understanding of the architecture and careful resource management by the software developer. Resource management that is tied to application development makes it difficult for packet processors to adapt to changes in the workload that are based on traffic conditions and the deployment of new functionality. Therefore, we present a network processor design that separates programming from resource management, which simplifies the software development process and improves the system's ability to adapt to network conditions. Based on our initial system design, we present a prototype implementation of a 4-core network processor using the NetFPGA platform. We demonstrate the operation of the system using header-processing and payload-processing applications. For packet forwarding, our simplified network processor can achieve a throughput of 2.79 Gigabits per second at a clock rate of only 62.5 MHz. Our results indicate the proposed design can scale to configurations with many more processors that operate at much higher clock rates and thus can achieve considerable higher throughput while using modest amounts of hardware resources.**

***Index Terms*—Router design, network processor, next-generation Internet, parallel processor, prototype**

I. Introduction

Modern routers use programmable packet processors on each port to implement packet forwarding and other advanced protocol functionality. This programmability in the data path is an important aspect of router designs in the current Internet – in contrast to the traditional approach where custom application-specific integrated circuits with fixed functionality are used. The ability to change a router's operation by simply changing the software processed on router ports makes it possible to introduce new functions (e.g., monitoring, accounting, anomaly detection, blocking, etc.) without changing router hardware. An essential requirement for these systems is the availability of a high-performance packet processor that can deliver packet processing at data rates of multiple Gigabits per second. Such network processors (NPs) have been developed and deployed over the last decade as systems-on-a-chip based on multi-core architectures.

One of the key challenges in using a network processor to implement advanced packet processing functionality is software development. Many programming environments for NPs use very low levels of abstractions. While this approach helps with achieving high throughput performance, it also poses considerable challenges to the software developer. Distributing processing workload between processor cores, coordinating shared resources, and manually allocating data structures to different memory types and banks is a difficult process. In environments where network functionality does not change frequently, it is conceivable to dedicate considerable resources to such software development. However, this software development approach becomes less practical in highly dynamic systems. The next-generation Internet is envisioned to be such a dynamic environment.

The next-generation Internet architecture is expected to rely on programmability in the infrastructure substrate to provide isolated network "slices" with functionally different protocol stacks [1], [2]. In such a network, the processing workload on router changes dynamically as slices are added or removed or as the amount of traffic within a slice changes. These dynamics require that the software on a network processor adapt at runtime without the involvement of a software developer. Thus, it is necessary to develop network processing systems where these dynamics can be handled by the system. The performance demands of packet processing do not allow the use of a completely general operating system. An operating system would use a considerable fraction of the network processor's resources. Instead, we focus on a solution where resource management is built into the network processor hardware and in turn allows a much simplified programming process.

The simplified network processor that we present in this paper attempts to hide the complexity of resource management in the network processor hardware. The software developer merely programs the functionality of packet processing. This approach contrasts other network processors, where functionality and resource management are tightly coupled (e.g., programmers need to explicitly choose allocation of data in SRAM or DRAM). By separating functionality from resource management, the system can more readily adapt to runtime conditions that could not have been predicted by the software developer.

We have described the general architecture of a simplified network processor in [3], [4], and we review the main aspects of the design in Section III. In this paper, we present a prototype implementation of our simplified network processor. Specifically, the contributions of our paper are:

This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-0626690 and CNS-0447873.

- FPGA-based prototype implementation of simplified network processor: To demonstrate the feasibility of our simplified network processor design, we present a 4-core prototype implementation based on the NetFPGA system [5]. This implementation shows that the proposed architecture can be realized with a moderate amount of resources.
- Functional operation with header-processing and payload-processing application: We present the results of two applications operating on the prototype system to demonstrate that the simplified network processor operates correctly and is able to process packet header and payloads. We also illustrate the simplicity of software development for the system.
- Performance results to demonstrate scalability: We present results on system throughput to show how well the system performs and how workload can be distributed over all processor cores. We also present results to indicate system scalability to larger number of cores and higher clock rates on ASIC-based implementations.

Overall, the results presented in this paper demonstrate that the simplified network processor architecture is feasible, efficient, and easier to program than conventional network processors. We believe that these results present an important step towards developing an efficient, easy-to-use infrastructure for packet processing in today's networks and the future Internet.

The remainder of this paper is organized as follows. Section II discusses related work. The overall system design of the simplified network processor is introduced in Section III. Specific details on the prototype implementation are presented in Section IV. Results from the prototype implementation and its performance are presented in Section V. Section VI summarizes and concludes this paper.

II. Related Work

Programmability in the data path of routers has been introduced as software extensions to workstation-based routers (e.g., Click modular router [6], dynamically extensible router [7]) as well as multi-core embedded network processors (e.g., Intel IXP platform [8], Cisco QuantumFlow processor [9], EZchip NP-3 [10], and AMCC nP series [11]). Programmability in the data path can be used to implement additional packet processing functions beyond simple IPv4-forwarding [12] or in-network data path service for next-generation networks [13], [14].

Software development environments for data path programming support general-purpose programmability [15], provide a modular structure (e.g., NP-Click [16], router plugins [17]), or implement abstraction layers to hide underlying hardware details [18]. While the management of packet I/O is largely simplified by software abstractions or libraries, control of packet transmission is still tied to packet processing in these approaches. Programming environments for network processors require software support for program partitioning (e.g., how to distribute workload over multiple processor cores) and resource management (e.g., how to allocate program state to different memories).

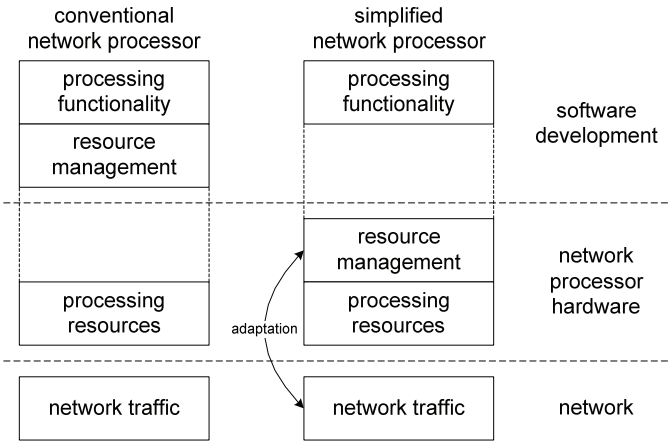

Fig. 1. Resource management in simplified network processor design.

In prior work, we have shown that it is possible to perform automated program partitioning of workloads [19] as well as to dynamically manage resources on multi-core network processors [20], [21]. Based on these results, we have proposed the basic architecture of the simplified network processor in [3], [4]. While we envision a specific implementation based on our prior work on workload partitioning and resource management, it is possible to use different runtime management approaches (e.g., [22], [23]).

III. System Architecture

We provide a brief overview on the system architecture of the simplified network processor before discussing details on the prototype implementation in Section IV. In principle, the network processor consists of a grid of packet processors that are locally connected to each other. A control system determines how packets are moved between processors and what processing is performed.

A. Resource Management

The system architecture of the simplified network processor is based on the idea of removing explicit resource management from the software development process and instead implementing this feature in the network processor hardware. Figure 1 shows this difference. The idea to not have the programmer handle resource management is not new – it has been very successfully deployed in practically any operating system. However, network processors (and many other embedded systems) do not use operating systems for reasons of performance. Network processors may use an embedded operating system on their control processor, but processors in the data – where performance really matters – are programmed directly.

Since network processors are used for a very specific task (i.e., processing packets), it is possible to provide resource management operations as part of the network processor system. Using a combination of special-purpose hardware resources and software on the control processor, it is possible to perform the following actions:

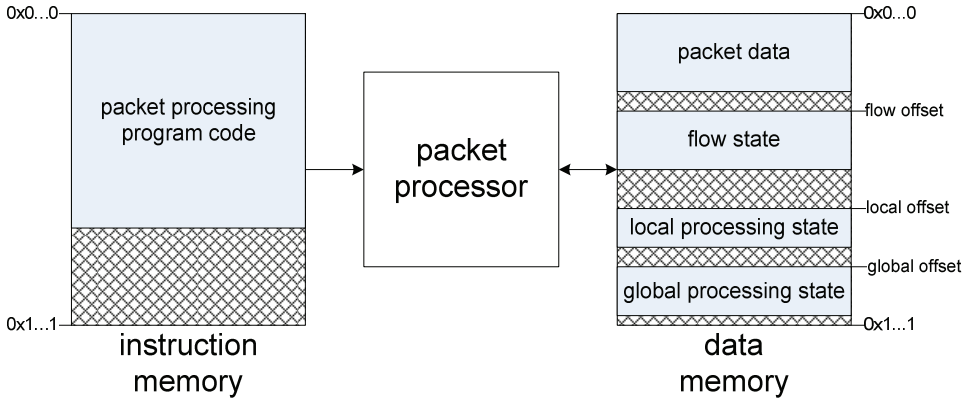

Fig. 2. Processing context in simplified network processor design.

- Move packets between processor cores,
- Switch processing context between different applications,
- Allocate processing resources to applications based on processing requirements, and
- Allocate memories to data structures based on access patterns.

We have discussed algorithms for the latter two functions in prior work (see [20] for dynamic mapping of tasks to processing resources and [21] for dynamic mapping of data structures to memories). These algorithms are implemented on the control processor of the system and thus are not immediately related to the network processor hardware. In this paper, we focus on the first two issues since they are an integral part of the hardware design.

B. Software Development

Using hardware support for moving packets between processor cores and switching processing contexts enables us to significantly simplify the software development process. Since the software developer does not need to explicitly manage packets or processing context, a much simpler processing environment can be presented. Through careful memory management, processing instructions, data structures, and the current packet can all be mapped to (virtually) fixed memory locations. Thus, the program can simply access them through static references. Figure 2 illustrates this simplified environment.

To make this approach practical, the network processor hardware needs to handle the necessary context switching and packet movement operations.

C. Packet Processor

The packet processor unit in the simplified network processor design is shown in Figure 3. The address shifters are used to map the most significant bits of memory addresses to the processing context that is currently in use by the processor. Similarly, access to the packet is mapped to one out of several in the packet buffer. For more details on the operation of the address shifter, see [3].

A critical aspect of the system is determining when to map memory accesses to which processing context. This step is handled by the resource management system on the control processor. Each packet is classified when entering the network processor to determine what processing steps are required (e.g., depending on the virtual slice to which a packet is sent). The packet then carries control information that determines its path and the processing steps that need be performed on different processors. Using this control information, each packet processor can determine what context needs to be mapped using the address shifters. Thus, it can be ensured that the correct processing steps are performed on the packet.

IV. Prototype Implementation

The high-level architecture of our four-core prototype system, which we have implemented on a NetFPGA [5], is shown in Figure 4. Packets enter through the I/O interface, get classified into flows and then distributed into the grid of packet processing units (PPUs). Each processing unit has a set of packet processing applications preloaded (as determined by the runtime system), and is able to select the requested application based on control information determined during packet classification. After the processing steps have been completed, the packet is sent through the output arbiter to the outgoing interface. The processor core is a 32-bit Plasma processor [24], which uses the MIPS instruction set and operates at 62.5MHz.

One of the key design aspects of this system is that packet processors only use local memory. Avoiding the use of a global, shared memory interface helps in preserving the scalability of the design. As we show in the results in Section V, the prototype implementation can scale to a 7×7 grid configuration with a linear increase in chip resources.

A. Packet Processing Unit

The setup of the packet buffer system is also illustrated in Figure 3. The packet buffers are used to store packets that are received from neighboring processor units (or from the flow classification unit). The processor can switch its local context to the packet that is being processed. Completed packets are stored until they can be passed to the neighboring processor units (or to the output arbiter). Bypass buffers are used for packets that do not need processing on the local processor, but need to be passed to a neighbor. By using

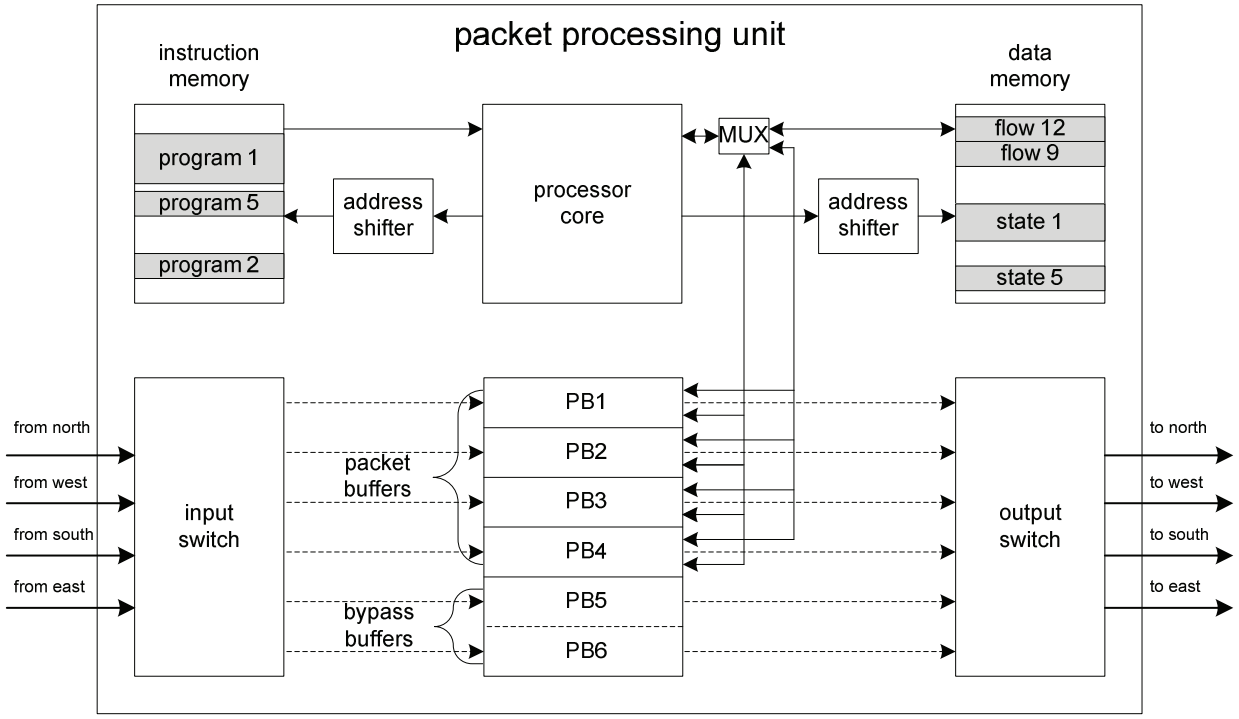

Fig. 3. Packet processing unit with support for context mapping and packet handling.

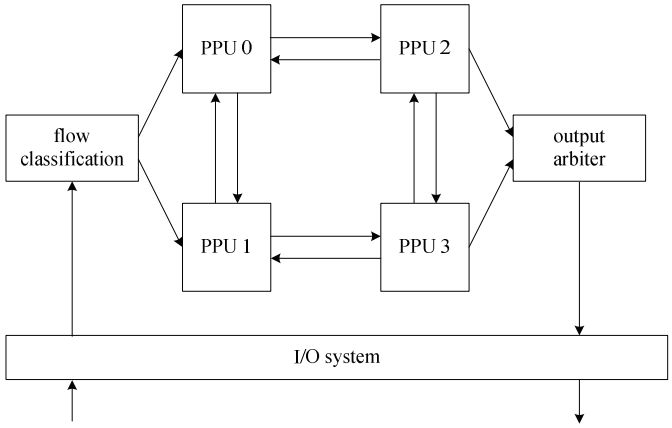

Fig. 4. Network processing platform design.

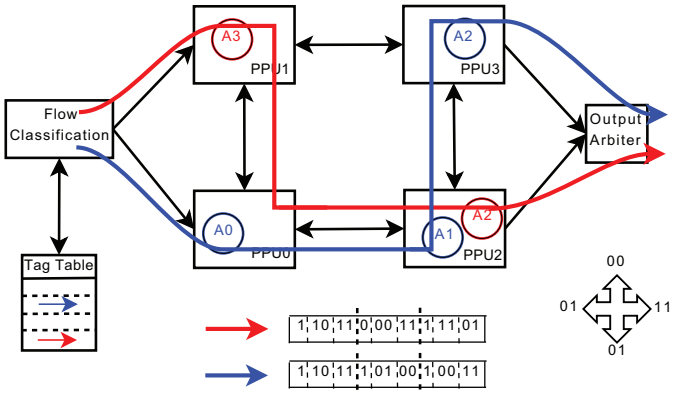

Fig. 5. Example of flow-based packet routing/processing and control information used by system.

separate buffers, blocking due to processor overload can be avoided. Our prototype uses four packet buffers for processing and two packet buffers for bypass. Larger numbers of buffers would help in reducing potential packet drops due to blocking, but limited on-chip memory resources impede larger buffer designs.

B. Flow Routing Mechanism

The flow classification unit of the system determines the transmission path of a packet through the system as well as the set of applications that are executed along the way. Each packet is augmented by control information that contains two pieces of information for each processor that is traversed in the grid:

- Service tag: The service tag consists of an indicator if an application is to be executed on the packet on the current processor. If processing is required, the additional bits in the service tag determine which application is used. In the current prototype, we support processing of one application per processor, but this design can be extended to support multiple applications per packet per processor.
- Routing information: The routing information indicates to which neighbor the packet should be passed after processing is completed.

Figure 5 shows an example of this control information for a 2×2 processor example with two active flows. The flows are routed as shown in the figure and processing occurs when the flow encounters an application illustrated by a circle. The tags that are kept in the tag table and that are added to the

packet are shown at the bottom of the figure. There are three triples of bit sequences. Each triple is used by one of the processors that are traversed. Note that the number of valid triples may change with different routes. Also, the triples are processed from right to left. Within a triple, the first bit indicates if an application is to operate on the packet. If so, the second bit sequence indicates the application identifier. The last bit sequence indicates the routing according to the directions shown in the lower right of the figure.

To setup (or change) the route of a flow or its processing steps, the runtime system of the network processor simply rewrites the control information in the tag table. This approach allows for very easy control of the system without the need to communicate with individual packet processing units.

Identification of flows is achieved through lookup operations on flow table stored in the classification unit. Thus, by altering entries of the flow table, a flow is able to access any service inside the processing grid. In addition, the bypass path of each PPU is isolated from the processing path to avoid blocking of bypass packet transmission. Thus, the flow routing mechanism allows for significant flexibility in the utilization of the processing grid. For example, all PPUs can be chained together to form a pipeline, or they can be logically parallelized (i.e., each flow can only be served by exactly one PPU). More details about application mapping on PPUs and the flow routing algorithm can be found in [25].

C. Simplified Programming Abstraction

As discussed in [3], [4], one of the goals of our design is to simplify code development for the network service processing platform. To achieve the desired simplicity, the packet processor is able to directly access on-chip memories, in which instructions (program code for multiple services), data and packets have been stored. As shown in Figure 2 the packet processor has an interface for reading program instructions and data memory and an interface for access to packet memory. In the instruction memory, the code for running a particular service is placed at a fixed, well-known offset. In the data memory we have placed the stack and global pointers at well-known offsets as well. With this design, packet processing and code development for packet processing is simplified. Packet data can be accessed via referencing the data memory on the (fixed) packet offset. Moreover, the program code is placed in a fixed location in the instruction memory and thus can be accessed easily by the processor.

An example of a piece of C code that accesses packet memory is shown in Figure 6. The code reads the time-to-live (TTL) field in the IP header and decrements it. Since the context is automatically mapped, the IP header can simply be accessed by a static reference. The hardware of the system ensures that this memory access is mapped to the correct physical address in the packet buffer that is currently in use. Similarly, data memory (and instruction memory) can be accessed. For example, to count the number of packets handled by an application, a simple counter can be declared:

```
static int packet_count
```

This counter can be incremented once per packet:

```
packet_count++
```

```
#define IP_TTL 0x1000001E
#define pkt_get8(addr, data) \
  data = *((volatile unsigned char *) addr)
#define pkt_put8(addr, data) \
  *((volatile unsigned char *) addr) = data

typedef unsigned char _u8;
_u8 ip_ttl;

pkt_get8(IP_TTL, ip_ttl);

if (ip_ttl != 0){
  ip_ttl--; \\decrement TTL
  pkt_put8(IP_TTL, ip_ttl);
} else {
  ...handle TTL expiration...
}
```

Fig. 6. Simple C program for accessing and decrementing the time-to-live field in the IP header.

The automated context handling ensures that the memory state is maintained for the application across packets, and thus correct counting is possible.

To program other network processors, a programmer has to specify the exact memory offset and memory bank (e.g., SRAM vs. DRAM) each and every time a data structure is accessed. Compared to this complex method of referencing memory, our programming model is considerably easier.

For our prototype implementation, we have implemented two specific applications:

- IP forwarding: This application implements IP forwarding [26] using a simple destination IP lookup algorithm.
- IPsec encryption: This application implements the cryptographic processing to encrypt IP headers and payload for VPN transmission [27].

These two applications represent two extremes in the spectrum of processing complexity. IP forwarding implements the minimum amount of processing that is necessary to forward a packet. IPsec is extremely processing-intensive since each byte of the packet has to be processed and since cryptographic processing is very compute-intensive.

V. Evaluation

In this section, we discuss performance results obtained from our prototype system. These results focus on functionality, throughput performance, and scalability.

A. Experimental Setup and Correctness

Using three of the Ethernet ports on the NetFPGA system, we connect the network processor to three workstation computers for traffic generation and trace collection. The routing and processing steps for flows on the network processor are set up statically for each experiment. The IP forwarding and IPsec application are instantiated as necessary on the processing units.

The first important result is that the system operates correctly. Using network monitoring on the workstation computers, we can verify that IP forwarding is implemented correctly

TABLE I
IP FORWARDING THROUGHPUT.

number of cores	small packets Mbps	small packets kpps	large packets Mbps	large packets kpps
1	154	302	2792	231
2	169	320	2769	229
3	167	327	2776	229
4	171	333	2785	230

TABLE II
IPSEC THROUGHPUT.

Number of cores	small packets Mbps	small packets kpps	large packets Mbps	large packets kpps
1	2.013	2.48	0.998	0.084
2	4.027	4.96	2.008	0.170
3	6.040	7.43	2.994	0.253
4	8.053	9.91	4.016	0.340

by the network processor (i.e., TTL decrement is performed correctly and the updated IP header checksum and link layer CRC are correct). This indicates that the software is correctly executed in our processors and that packets are passed through the system correctly.

B. Data Path Throughput

The throughput results for IP forwarding demonstrate the overall throughput performance of the prototype system. Table I shows the forwarding performance for small packets (64 bytes) and large packet (1512 bytes). For large packets, our network processor can achieve a peak forwarding rate of 2.79 Gigabits per second with only a single core. For small packets, the forwarding rate drops due to the per-packet overhead.

Note that our prototype is built on an FPGA-based system, which is convenient for prototyping, but not realistic for deployment in a real network. The clock rate of the processor units is only 62.5MHz. Achieving several Gigabits per second forwarding performance on such a system is a considerable achievement. In a realistic deployment, an ASIC-based implementation with considerably higher clock rates would be used. Thus, the forwarding performance would scale up accordingly.

C. Processing Capacity

To evaluate the processing performance of the system, we consider the processing-intensive IPsec application. Table II shows IPsec performance for small and large packets. The forwarding performance for small packets is about twice that of large packets since small packets contain proportionally more header information that does not need to be encrypted. While the overall data rate is low (as expected), it is important to note that the performance scales linearly with the number of processor cores. Thus, we can see that the system design scales without creating bottlenecks when using multiple processors – even though all processors are operating at maximum load. In particular, the flow routing system can isolate packet handling from processing. Again, transferring the system to an ASIC-based implementation would improve processing performance considerably.

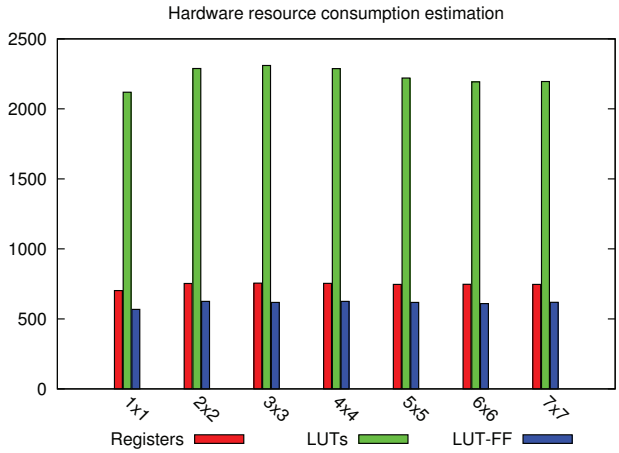

Fig. 7. Resource estimation per processing unit on Virtex 5.

D. Hardware Resource Utilization

To evaluate the scalability of our system architecture, we synthesized different configurations of our design on a larger FPGA (Virtex 5 XC5VLX330T). Table III shows the hardware resource consumption for 5 different processing grid configurations. The increase in resource consumption scales with the number of processing units. The amount of registers, lookup tables, and flip-flops *per processing unit* is shown in Figure 7. The nearly constant values across all configurations indicate that our system can be scaled to large configuration without bottlenecks.

Table III also provides a rough estimate of the performance using the maximum operating clock rate reported by Xilinx synthesis tool. As we use on-chip block ram to work with the Plasma core in prototype system, throughput of processor pipeline is approximately 1 CPI. We assume that an application executes an average of 500 instructions per packet (which is more than is necessary for forwarding) and average packet sizes of 300 bytes. Based on the results in Table III, we can see that system performance scales to tens of Gigabits per second.

Thus, we believe that our simplified network processor system can provide ease of use as well as the necessary performance to support packet processing in the networks of today and the future.

VI. SUMMARY AND CONCLUSIONS

In this paper, we present a novel network processor design that completely separates resource management from packet processing application development. Thus it provides a simple programming abstraction to software developers and maintains high scalability as the architecture moves towards large number of integrated packet processing cores. The 4-core prototype system built on NetFPGA platform can achieve 2.79 Gbps of forwarding throughput at processing clock rate of only 62.5 MHz. Performance estimate indicates that the architecture could support tens of Gigabit per second rate with modest hardware resource utilization.

TABLE III
RESOURCE CONSUMPTION AND PERFORMANCE ESTIMATION.

Processing grid configuration	Slice registers	Slice LUTs	Fully used LUT-FF pairs	Maximum Frequency (MHz)	Aggregated processing capacity (MIPS)	Estimated maximum throughput (Gbps)
3×3	6,795	20,791	5,561	158.644	1,427.796	6.853
4×4	12,052	36,601	9,999	161.180	2,578.880	12.379
5×5	18,662	55,516	15,442	131.578	3,289.450	15.789
6×6	26,892	78,956	21,919	147.626	5,314.536	25.510
7×7	36,554	107,592	30,300	129.735	6,357.015	30.514

REFERENCES

[1] T. Anderson, L. Peterson, S. Shenker, and J. Turner, "Overcoming the Internet impasse through virtualization," *Computer*, vol. 38, no. 4, pp. 34–41, Apr. 2005.

[2] J. S. Turner and D. E. Taylor, "Diversifying the Internet," in *Proc. of IEEE Global Communications Conference (GLOBECOM)*, vol. 2, Saint Louis, MO, Nov. 2005.

[3] Q. Wu and T. Wolf, "Design of a network service processing platform for data path customization," in *Proc. of The Second ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of TOmorrow (PRESTO)*, Barcelona, Spain, Aug. 2009, pp. 31–36.

[4] Q. Wu, D. Chasaki, and T. Wolf, "Simplifying data path processing in next-generation routers," in *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)*, Princeton, NJ, Oct. 2009.

[5] 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 '07: Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, San Diego, CA, Jun. 2007, pp. 160–161.

[6] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click modular router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[7] F. Kuhns, J. DeHart, A. Kantawala, R. Keller, J. Lockwood, P. Pappu, D. Richard, D. E. Taylor, J. Parwatikar, E. Spitznagel, J. Turner, and K. Wong, "Design of a high performance dynamically extensible router," in *Proc. of DARPA Active Networks Conference and Exhibition*, San Francisco, CA, May 2002.

[8] *Intel Second Generation Network Processor*, Intel Corporation, 2005, http://www.intel.com/design/network/products/npfamily/.

[9] *The Cisco QuantumFlow Processor: Ciscos Next Generation Network Processor*, Cisco Systems, Inc., San Jose, CA, Feb. 2008.

[10] *NP-3 – 30-Gigabit Network Processor with Integrated Traffic Management*, EZchip Technologies Ltd., Yokneam, Israel, May 2007, http://www.ezchip.com/.

[11] *np7300 10 Gbps Network Processor*, AMCC, 2006, http://www.amcc.com.

[12] W. Eatherton, "The push of network processing to the top of the pyramid," in *Keynote Presentation at ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)*, Princeton, NJ, Oct. 2005.

[13] T. Wolf, "Service-centric end-to-end abstractions in next-generation networks," in *Proc. of Fifteenth IEEE International Conference on Computer Communications and Networks (ICCCN)*, Arlington, VA, Oct. 2006, pp. 79–86.

[14] S. Ganapathy and T. Wolf, "Design of a network service architecture," in *Proc. of Sixteenth IEEE International Conference on Computer Communications and Networks (ICCCN)*, Honolulu, HI, Aug. 2007, pp. 754–759.

[15] S. D. Goglin, D. Hooper, A. Kumar, and R. Yavatkar, "Advanced software framework, tools, and languages for the IXP family," *Intel Technology Journal*, vol. 7, no. 4, pp. 64–76, Nov. 2003.

[16] N. Shah, W. Plishker, K. Ravindran, and K. Keutzer, "NP-Click: A productive software development approach for network processors," *IEEE Micro*, vol. 24, no. 5, pp. 45–54, Sep. 2004.

[17] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner, "Router Plugins - a modular and extensible software framework for modern high performance integrated services routers," in *Proc. of ACM SIGCOMM 98*, Vancouver, BC, Sep. 1998, pp. 229–240.

[18] *nPsoft Development Environment Product Brief*, AMCC, 2006, http://www.appliedmicro.com.

[19] N. Weng and T. Wolf, "Analytic modeling of network processors for parallel workload mapping," *ACM Transactions on Embedded Computing Systems*, vol. 8, no. 3, pp. 1–29, Apr. 2009.

[20] Q. Wu and T. Wolf, "On runtime management in multi-core packet processing systems," in *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)*, San Jose, CA, Nov. 2008, pp. 69–78.

[21] ——, "Runtime resource allocation in multi-core packet processing systems," in *Proc. of IEEE Workshop on High Performance Switching and Routing (HPSR)*, Paris, France, Jun. 2009.

[22] R. Kokku, T. Riché, A. Kunze, J. Mudigonda, J. Jason, and H. Vin, "A case for run-time adaptation in packet processing systems," in *Proc. of the 2nd Workshop on Hot Topics in Networks (HOTNETS-II)*, Cambridge, MA, Nov. 2003.

[23] T. Wolf, N. Weng, and C.-H. Tai, "Run-time support for multi-core packet processing systems," *IEEE Network*, vol. 21, no. 4, pp. 29–37, Jul. 2007.

[24] S. Rhoads, *Plasma – most MIPS I(TM) Opcodes*, 2001, http://www.opencores.org/project,plasma.

[25] Q. Wu and T. Wolf, "Data path management in mesh-based programmable routers," in *Proc. of IEEE International Conference on Communications (ICC)*, Cape Town, South Africa, May 2010.

[26] F. Baker, "Requirements for IP version 4 routers," Network Working Group, RFC 1812, Jun. 1995.

[27] S. Kent and R. Atkinson, "Security architecture for the Internet protocol," Network Working Group, RFC 2401, Nov. 1998.