Simplifying Data Path Processing in Next-Generation Routers

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

Customizable packet processing is an important aspect of next-generation networks. Packet processing architectures using multi-core systems on a chip can be difficult to program. In our work, we propose a new packet processor design that simplifies packet processing by managing packet contexts in hardware. We show how such a design scales to large systems. Our results also show that the management of such a system is feasible with the proposed mapping algorithm.

General Terms

Design, Performance, Algorithms

Categories and Subject Descriptors

C.2.6 [**Computer-Communication Networks**]: Internetworking—*Routers*; C.1.4 [**Processor Architectures**]: Parallel Architectures

1. INTRODUCTION

Routers in the Internet perform packet processing functions not only to implement standard network-layer protocol operations, but also an increasingly diverse set of security, quality-of-service, traffic management, accounting, and measurement functions [4]. To implement these operations, and to support new features as they emerge, the data path of modern routers is implemented using programmable packet processing systems (e.g., [2]). By adapting the software that is executed on these packet processors, the functionality of the router (and ultimately the network) can be changed.

This trend towards software-based network systems can also be observed in academic research, where efforts are focused on defining the next-generation network architecture [5]. Many of the proposed architectures use programmability in the data path in one of two ways: (1) programmability as a way of deploying new protocols in the network infrastructure, or (2) programmability as an explicit feature of the network architecture. The prior use of programmability is particularly important for architectures that are based on network virtualization [1], where slices with new protocols stacks are deployed dynamically during the operation of the network. The latter use of programmability exposes some of the processing capabilities of the infrastructure to allow the dynamic deployment of new functionality within a network or a slice (e.g., network services [6]). In either scenario, a high-performance programmable packet processing system is at the core of router design.

Programmable packet processing engines can be implemented using several different technologies ranging from general-purpose workstations processors [3] to embedded multicore systems-on-a-chip network processors [18] and programmable logic devices [19]. One of the dominating design tradeoffs in this space is the tension between processing performance and simplicity of use (i.e., ease of programming). Workstation processors provide an environment where code development is easy due to significant support by the operating system and established development tools (e.g., Click [11]). However, the raw processing performance of network processors and programmable logic devices is higher. In contrast, these systems do not use operating systems and thus require more complex code development.

In our work, we focus on the design of a packet processing system that aims to provide high performance as well as ease of use. The main idea is to handle the context management of instructions, packets, processing and flow state in hardware. This hardware management simplifies the way packet processing functions can access relevant data thus significantly simplifying code development. Specifically, the contributions of the research described in this paper are:

- An overview on the design of a packet processing system that provides a simple programming interface while hiding the complexities of context management. The challenge in this aspect of the work is to design the processing engines in such a way that instructions, flow state, program memory, and packet memory can be handled automatically. The processor presents a simple single-core programming environment with static memory spaces to the application to allow for simple code development.
- The results of a design space exploration that highlights the scalability of the system design to support large numbers of active flows with diverse processing needs. We use synthesis results from a configurable processor core implementation to estimate the area requirements and processing performance of different

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ANCS'09, October 19–20, 2009, Princeton, New Jersey, USA.
Copyright 2009 ACM 978-1-60558-630-4/09/0010 ...$5.00.

system configurations. The results quantify the capabilities of different system configurations to support large numbers of flows.

- An algorithm for runtime task allocation in our highly parallel packet processing system. The challenge in managing the processing resource in the architecture that we describe is to ensure that a centralized control system sets up processing contexts and interconnections correctly for all flows. We use profiling information to estimate the processing requirements of the system. A mapping algorithm dynamically allocates flows and packet processing tasks to system resources at runtime.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 presents an overview of our packet processing system architecture. A quantitative evaluation of the design space is presented in Section 4. Section 5 presents a discussion of runtime management issues including a mapping algorithm. Section 6 summarizes and concludes this paper.

2. RELATED WORK

Customization in data path processing has been studied as an aspect of networking systems in form of extensible operating system kernels (e.g., x-Kernel [9]) and routers with selectable features (e.g., router plugins [3]). More general programmability has been proposed for active network [20], network virtualization [1], and network services [6].

Several implementations of programmable packet processing systems are based on conventional workstation processors (e.g., Click modular router, [11], dynamically extensible router [13]). More recently, multi-core embedded network processors (e.g., Intel IXP platform [10]) have been used for router designs [18, 21]. In such systems, it is important to consider programming abstractions (e.g., NP-Click [17], advanced compilers and runtime systems [7]) that can be used to simplify software development. These development environments support offline profiling and optimization, but do not support dynamic adaptation to changing traffic workloads. Runtime support systems that manage processing resources have been proposed to support more dynamic environments [12,22,23]. Our work differs from these approaches as it proposed hardware support for context management (which can be seen as a form of runtime management).

The idea of using a larger number of simpler processors to achieve high throughput on highly parallel workloads is not new. Single instruction multiple data (SIMD) computer architectures have used this approach. Graphics processing units (GPU) are a very successful example of highly parallel processors. However, their use in networking is limited since the workloads of network processing do not match the SIMD processing model.

Adapting processor design for the networking domain has been explored in a variety of directions. For example, instruction sets have been extended to support networking tasks [8]. Network processors designs that provide hardware support to perform multiple protocol-processing functions have been proposed in [15]. The focus of our work is to further improve hardware support and to further simplify process. Also, support for run-time adaptation of workloads is a critical aspect of our work.

3. SYSTEM ARCHITECTURE

The main concepts of the design of our packet processing system are discussed in this section. More details on design tradeoffs and scalability are covered in Section 4. Issues on runtime management are discussed in Section 5. Some initial ideas on this system have been described in [24].

3.1 Simplicity in Data Path Processing

The main design goal of our system is to provide a simple packet processing system that can be programmed easily and achieves high throughput. The goal of simplicity in the data path follows two dimensions:

- Simplicity in processor design.

- Simplicity through hardware context management.

3.1.1 Simple Processor Design

When designing a packet processing system, a design choice needs to be made on how complex a processor is used at the core of the system. More complex processors typically perform processing faster. However, they also require more hardware resources and consume more power. Assuming a fixed resource budget, a design can use either a larger number of simpler processor or a smaller number of larger processors. Using a larger number of simpler processor for packet processors provides several advantages specific to the networking domain:

- Higher throughput: Network processors are concerned with the throughput of the system (i.e., total amount of processing across all processors) rather than per-packet delay (i.e., executing speed of single processing task). A larger number of lower-performing processors achieves an overall higher throughput since the cost for speeding up single-thread performance is disproportionate to the performance gain. (Note that the processing delay on a router – even when performing complex tasks on a slow processor – is typically considerably less than the propagation delay encountered on transmission links. Thus, increasing delay for higher throughput is acceptable.)

- Lower power consumption: Following the argument for higher throughput, a system with more lower-speed processors consumes less power while achieving higher processing performance.

- Easier support for virtualization: It is convenient to provide isolation between slices at the level of a processor rather than attempting to achieve isolation of processing tasks within a processor. Having a larger number of processors available provides the basis for finer-grained resource allocation than is possible with a few large processors.

However, there are also several challenges when using a larger number of simpler processors. In particular, the management of these resources becomes more complex. We present some solutions to this problem in Section 5. Nevertheless, using a larger number of smaller processors is preferable when considering performance.

3.1.2 *Hardware Context Management*

One of the challenges in a packet processing system is to ensure that each packet processing function has access to data it needs. There are different types of data and information that are used for network processing:

- Packet processing program code: The instructions for processing a packet are typically stored in a dedicated instruction store.
- Packet data: The packet header and often the packet payload needs to be accessible to the packet processing function.
- Flow state: Many packet processing functions are not entirely stateless but maintain a small amount of per-flow state.
- Local processing state: This state encompasses temporary memory used by the packet processing function.
- Global processing state: This state information encompasses global per-function state as well as system-wide global state.

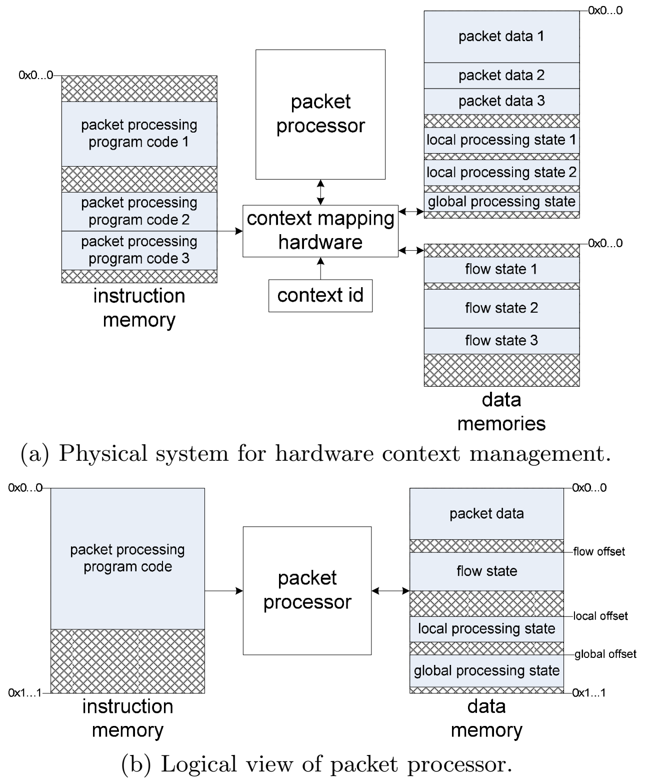
We use the term "context" to refer to the set of all these data and state information that are specific to one particular packet. In a typical implementation of packet processing systems, a processing functions that tries to access this context needs to explicitly find the correct memory locations where it is stored. Such a manual management of packet processing context can lead to difficulties when programming packet processing functions and can also consume considerable processing resources. Instead, our packet processor design is based on providing hardware context management.

3.2 Packet Processor Design

The most important aspect of our packet processor design is the way access to packet context is handled by the system. The key idea is shown in Figure 1. Figure 1(a) shows the physical system, where multiple instances of context exist in possibly multiple different physical memories. Using the context mapping hardware, the system can "switch" between different context based on the context identifier. The logical view of what is visible to the packet processor is shown in Figure 1(b). All relevant data are directly accessible at fixed memory offsets. Thus, performing packet processing functions becomes very simple (as does code development).

The detailed design of the packet processor is shown in Figure 2. Context management is achieved using an address shifter. The address shifter component is responsible for mapping the processor core's address space to the memory section that contains the appropriate data. We discuss this functionality in the context of instruction memory, but it also applies to data memory as shown in Figure 2.

In our example, we assume a 16-bit address space used by the processor core. Note that this design can also be parameterized for other memory space sizes. We only present one configuration for illustrative purposes. On the interface between the packet processor and the instruction memory, the least significant bits (the lower eight bits in the system shown in Figure 2) are connected directly to the instruction memory. Thus, the address translation operates in chunks of 256 bytes. The eight most significant bits are sent into an 8-bit address shifter component.



(a) Physical system for hardware context management.

(b) Logical view of packet processor.

Figure 1: Hardware context management present relevant context to packet processing in simplified address space.

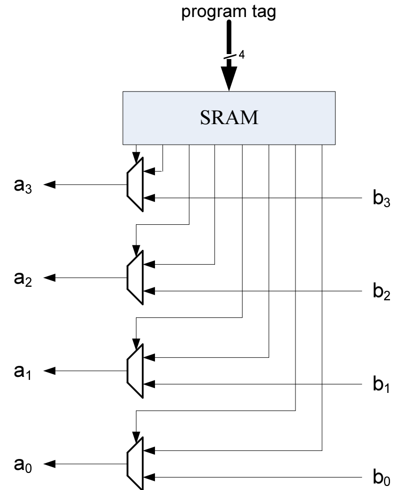


Figure 3: Address Shifter Design.

The design of a 4-bit address shifter is shown in Figure 3. By sending the appropriate control signals from SRAM, the address shifter has the ability of either (1) forward an address bit (i.e., outputting b_i on a_i) or (2) overwrite an address bit (i.e., outputting a stored value from SRAM on a_i). With these two options, the packet processor core's address space can be shifted to another range in the address space and it can be limited in size (if less than 64kB

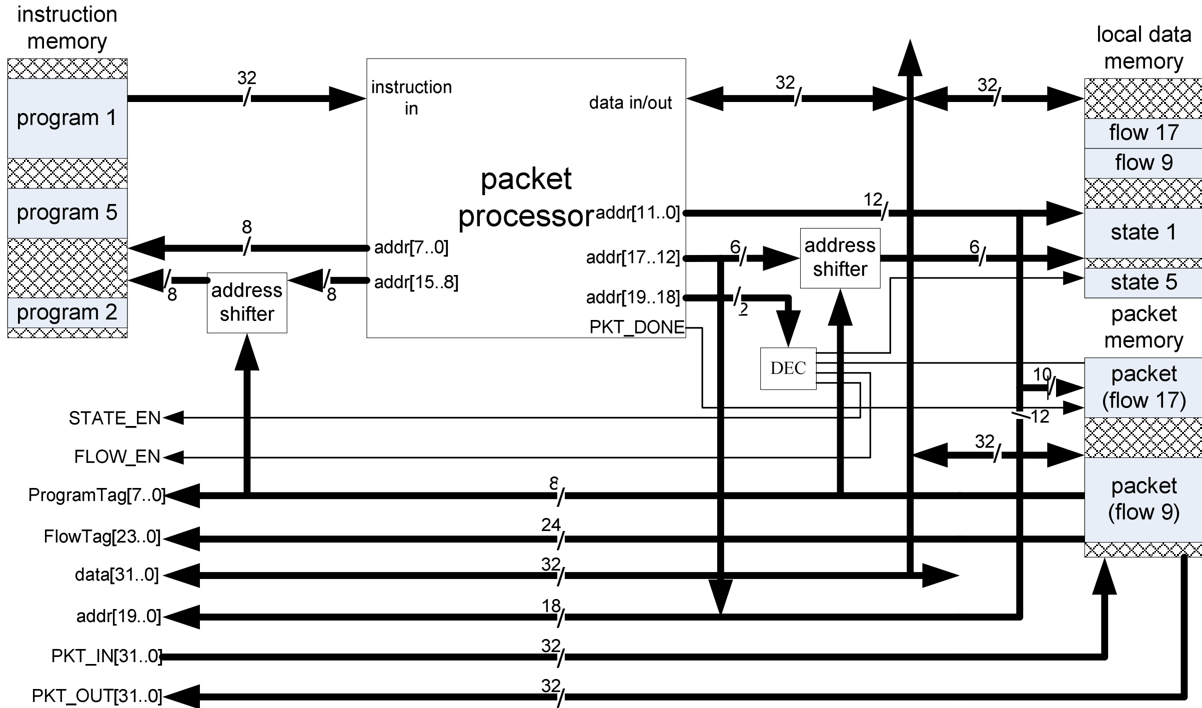


Figure 2: Packet Processor Design.

are needed). For example, the processor uses a 10-bit address space, which is stored in physical memory from `0x0400` to `0x07FF`, address bits 0...7 are connected directly to the memory, address bits 8...9 are forwarded by the address shifter, address bits 10 is set to 1 by the address shifter, and all higher address bits are set to 0. Note that if the overall address space for programs needs to exceed 16 bits, additional address lines can be controlled by the SRAM in Figure 3 (without having a choice of forwarding processor address lines since they do not exist beyond 16 bits). The selection of signals sent to the multiplexers depends on the *program tag*, which is part of the context identified.

Each packet also carries context that ensures that it is processed correctly. This information consists of:

- Program tag: This tag identifies which packet processing function is to be performed by which packet processor.
- Flow tag: The flow tag identifies to which flow a packet belongs. This tag is used to demultiplex to the correct flow state information.

These tags need to be set correctly before initiating the processing of a packet. If multiple packet processing functions are to be performed on a packet, multiple program tags are necessary. This setup of tags and corresponding contexts is performed by the control system.

3.3 System Design

The system-level design of the packet processor system is shown in Figure 4. Packets that arrive in the system are classified and demultiplexed into the grid of packet processors. By utilizing the inherent inter-packet parallelism of network traffic, flow classification and packet distribution can be parallelized and scaled to the necessary throughput

Table 1: Memory configurations.

	Instruction memory memory	Packet memory	Data memory
Conf. 1	64K	16K	16K
Conf. 2	64K	16K	64K
Conf. 3	64K	16K	256K

performance. Using the program and flow tags, each processor in the grid can determine what packet processing function to perform on the packet and where to send the packet next.

The control of this system is performed through the setup of program and flow states on the processors. (Note a global shared memory is not shown in Figure 4.) Using this setup capability, the system can place processing tasks onto processors and assign the paths taken by packets belonging to particular flows. More about this management aspect is presented in Section 5.

4. DESIGN RESOURCE REQUIREMENTS

In this section, we evaluate the packet processor system design that was described in Section 3 using the soft processor environment SPREE (Soft Processor Rapid Exploration Environment). SPREE allows us to generate large soft-processor designs and target commercial FPGA and ASIC devices. A set of processors has already been generated and verified by SPREE at the University of Toronto. Under the assumption that our packet processor uses a similar, MIPS-based core, we synthesize and simulate the pipelined architectures in order to estimate the size and the clock rate of the system design.

Since SPREE processors are compatible with Altera hard-

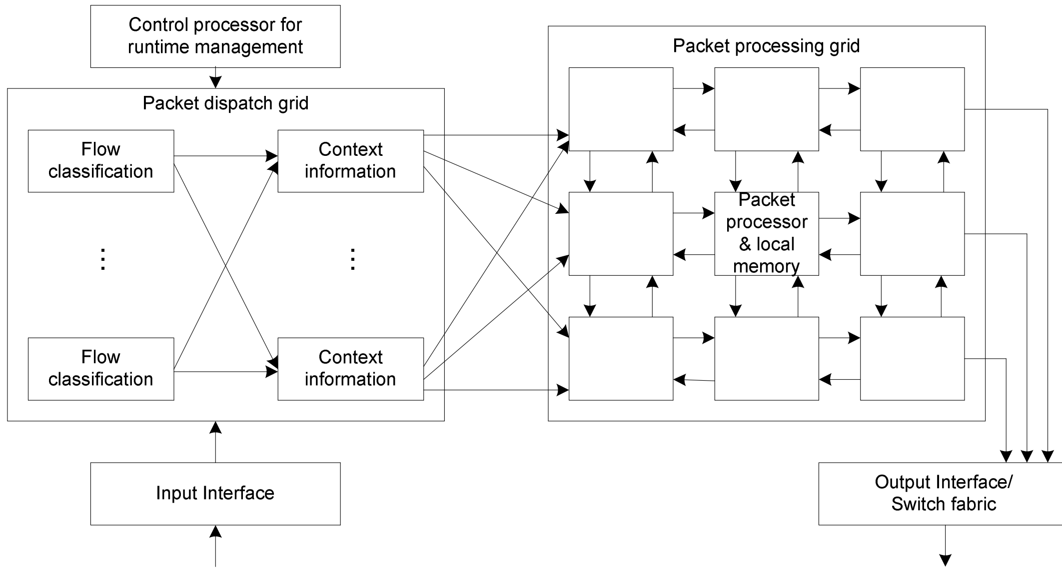


Figure 4: Packet Processor System Design.

Table 2: Resource requirements on FPGA.

FPGA size (ALUTs/Memory bits)	Conf. 1	Conf. 2	Conf. 3
8-bit	549/2,361,344	556/2,754,560	690/4,327,424
16-bit	576/2,623,488	587/3,409,920	768/6,555,648
32-bit	619/3,147,776	655/4,720,640	900/11,012,096

ware, we use Altera Quartus II as our synthesis tool, and Modelsim as the simulation tool. Several 32-bit processor systems are available in SPREE - implemented with three, five and seven pipeline stages. We design and synthesize 8 and 16 bit width processors as well, in order to obtain a fair view of the size requirements of both smaller and larger single processors. The target devices are Startix II FPGAs and Hardcopy II ASICs. Although Stratix II hardware is used in the SPREE flow as a single processor, SPREE also has the potential to program multiple processors. Based on the size that is required for one service processor we estimate how many processors we can afford to have, so that our system supports the desired amount of services and flows. Moreover, we draw conclusions about the width size of the processor that best suits our design.

Stratix devices offer up to 15Mbits of on-chip memory while an average Stratix-II FPGA has around 100,000 adaptive LUTs. Hardcopy II Asic devices support approximately 9 Mbits of memory and up to 3,000,000 Hcells. Table 2 shows the size required on an FPGA device to build a service processor of 8-16 or 32 bits for different memory configurations. We assume scenarios where we would need 64K of instruction memory and 16-64 or 256K of data memory for a single service processor. In Table 3, we show the respective size requirements when we target an ASIC device.

The bottleneck of the processor architecture is not the number of hcells that are needed to build the functional units of the processor, but the instruction and data memory. With Hardcopy IV series being in the 40-nm scale, current ASICs already contain 2.8M to 15M usable gates and up to 20.3 Mbits of on-chip memory. These figures are going to rise even more in the near future. Taking into account the maximum size of commercial ASICs we calculate the number of processors that can potentially fit in them. Assuming - for simplicity - that we have one processing step per flow we can estimate the state per flow that we need for our service processor. Figure 5 depicts the state/flow for the different number of flows that the system supports. We can see how the state varies for several memory configurations.

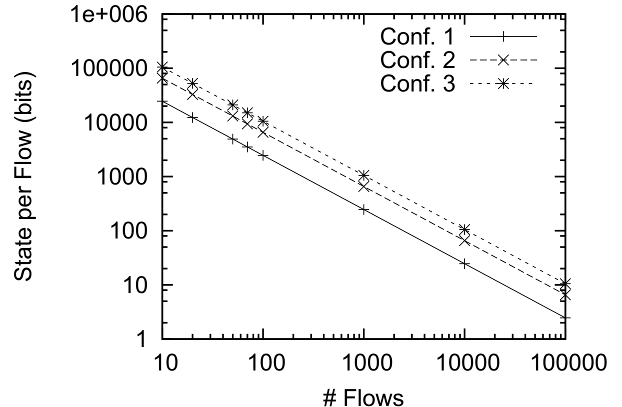


Figure 5: Available flow state per flow.

It is also important to identify the speed at which we can have the service processor running. During synthesis, we enable the speed optimization option, so that the fitter makes the highest effort even after meeting the timing re-

Table 3: Resource requirements on ASIC. The two largest designs could not be synthesized due to limitations in the development tools.

ASIC size (Hcells/Memory bits)	Conf. 1	Conf. 2	Conf. 3
8-bit	16,858/2,361,344	16,954/2,754,560	17,491/4,327,424
16-bit	16,950/2,623,488	17,249/3,409,920	–/6555648
32-bit	17,071/3,147,776	18,124/4,720,640	–/11,012,096

Table 4: Maximum clock frequency.

Max clock rate	8-bit	16-bit	32-bit
Pipeline 3	139.28 MHz	137.84 MHz	128.32 MHz
Pipeline 5	141.66 MHz	138.62 MHz	134.93 MHz
Pipeline 7	178.16 MHz	174.67 MHz	163.24 MHz

quirements. Table 4 shows the maximum clock rate in which the service processor can function. We observe that an 8-bit processor with seven pipeline stages gives the fastest results.

Overall, these results show that our design can support a large number of packet processors on ASICs as well as on experimental FPGA platforms (e.g., NetFPGA [14]).

5. RUNTIME MANAGEMENT

The packet processing system above requires an effective control system to set up hardware context and manage resources during runtime. In environments where packets from different flows require different types of processing, a static allocation can not handle changes in traffic patterns (unless over-provisioning is possible, which it is not in most systems). Therefore, a runtime system needs to perform the following functions:

- Mapping of packet processing functions to processors. This step require placement of context information in each processor's local memory.
- Monitoring and profiling of actual workload. During runtime, the system needs to keep track of what resources are being used based on current traffic load.
- Dynamic adaptation of resource allocation. As traffic requirements change, the packet processor needs to adapt the resource

Runtime systems have been designed for network processors [12, 22, 23]. We follow some of the ideas in [23] on how to handle dynamic task mapping in our system. In particular, we focus on the mapping algorithm that places packet processing functions on packet processors in our system. To support runtime adaptation, this algorithm can simply be periodically to update the system.

5.1 Task Mapping Algorithm

The task mapping algorithm designed for our system consists to three steps:

1. Workload profiling: Flow statistics are collected from packet I/O subsystem to derive task utilization and edge utilization.
2. Task replication: Tasks that are processing intensive and are used heavily are replicated across multiple processors to increase the processing resources dedicated to them. This step determines which tasks need to be replicated.
3. Task and edge mapping: The final step places tasks on processors while considering the limits on the bandwidth of the interconnect.

We discuss each step in more detail.

5.1.1 *Workload Profiling*

The task mapping algorithm requires knowledge of the workload of the system, i.e., how much traffic requires which packet processing function(s). Therefore, it is important to have some profiling information about the workload.

The flow of all packets is represented by a directed graph, where packet processing functions are nodes and traffic traverses edges. The characteristics of this graph are determined by the type of traffic that is sent via the router on which the packet processing system is installed. To determine the processing requirements, we collect two metrics: utilization and processing times. For utilization, we profile the utilization $u(t_i)$ of each task t_i and the utilization $u(e_{i,j})$ of each edge $e_{i,j}$. These utilizations values indicate how frequently a tasks is used and how much traffic is sent between tasks. The processing time s_i of task t_i captures the processing complexity of the task. In practice, processing times vary between packets and s_i represents an average value. With this profiling information, we express the total system processing demand, P_{sys} as $P_{sys} = \sum_i s_i \times u(t_i)$.

5.1.2 *Task Replication*

Task replication is a technique to balance the workload of tasks in a network processing system [23]. The main idea is to create multiple instances of tasks that require the largest amount of work. Work, w_i, of task t_i is defined as the product of utilization and processing time: $w_i = u(t_i) \times s_i$. Note that high work can be caused by high utilization (and possibly simple processing) or high processing complexity (and possibly low utilization).

A task that requires high work, is split into multiple instances (i.e., copies). Traffic is distributed evenly between the copies. As a result, the effective utilization for each copy decreases by a factor d if the tasks is duplicated d times. Therefore, the work for each task instance is reduced.

Using a simply greedy algorithm that iteratively duplicates the most work-intensive task achieves a very balanced workload. Note that the duplication process terminates

when there are as many tasks as can be installed on the system. In addition, note that a beneficial side effect of duplication is the creation of many simple tasks, which is important for our packet processing system, where a large number of simple processors need to be utilized to achieve high throughput.

5.1.3 *Task and Edge Mapping*

Given a graph of processing tasks (and their duplicates), the mapping algorithm needs to place the tasks on N processor cores with M hardware threads in each core. Since tasks are nearly balanced in the amount of work that they perform (due to task replication as explained above), there are no bottleneck tasks that can hold up the flow of packets through the system. Therefore, it is not necessary to solve a packing problem, where work-intensive tasks need to be co-located with simple tasks to avoid overloading of a processor. Instead, the algorithm can simply place an equal number of tasks on all processors and focus on reducing the interconnect load.

Algorithm 1 shows how the mapping m is computed. This algorithm requires all inter-processor connection capacities $C_{x,y}$ (from processor x to y) to be initialized before mapping starts. For processors that are not directly connected, the capacity is set to zero. Mapping starts with placing the first task, which is assumed to be the ingress node of all traffic, to an empty thread on the first processor. Then, the map_next function locates the edge with highest utilization among all outgoing edges of the latest mapped task. If the current processor that hosts the latest mapped task still has an empty thread and the remaining capacity of connection from the processor's output to its input is enough to support the edge utilization, the task that is pointed to by this edge is mapped to current processor. Otherwise, the algorithm tries to map it to the neighbors of current processor. If all steps succeed, this process is repeated recursively to achieve a depth-first mapping. The recursion terminates when a task has no outgoing edge to unmapped tasks (e.g., egress task). It is possible that at some point during mapping, the current processor and all its neighbors fail to meet the requirement of both empty thread and enough interconnection capacity. This indicates the system processing demand P_{sys} exceeds system capacity. In this case, P_{sys} needs to be adjusted (e.g. dropping a flow at the system input interface) and mapping needs to be repeated.

5.2 Evaluation

To show the effectiveness of the mapping algorithm, we show several evaluation results. We use a simulation based on PacketBench [16] using seven representative packet processing functions including packet classification, IPv4 forwarding (LC trie and radix tree), string matching, IPsec encryption and decryption , and QoS for workload evaluation. These packet processing functions are partitioned into a task graph representation comprised of 25 tasks and 29 edges. We evaluate the quality of mapping for different hardware configurations (i.e., different number of processors and threads per processor).

As our packet processing architecture focuses on computational capacity by deploying more processors, it is important to understand the effect of increasing number of processors on overall system workload. Figure 6 shows results of average processor utilization, which indicates the balance of workload among processors. In general, higher utilization implies less idle time for processors, thus leads to higher throughput as processors devote more time to packet processing. It can be observed from Figure 6 that processors get better utilization when more processors are present in system. Due to more effective task duplication, the workload is spread more evenly across processors. In addition, average utilization is typically better with more threads per processor.

System performance is also determined by inter-processor communication. With insufficient capacity on the interconnect, the system encounters a bottleneck. Therefore we explore the need for inter-processor communication in Fig-

Algorithm 1 Resource Mapping Algorithm.

Require: $C_{x,y} \mid x,y \in 1 \ldots N$ is initialized
1: **function** map_next(i,p)
2: **while** $\exists\ e_{i,j}$ with t_j unmapped **do**
3: $k \leftarrow \text{argmax}_j(u(e_{i,j}))$
4: **if** tasks_allocated_to(p) $\leq M$ and $C_{p,p} \geq u(e_{i,j})$ **then**
5: $m(t_k) \leftarrow p$
6: map_next(k,p)
7: $C_{p,p} \leftarrow C_{p,p} - u(e_{i,j})$
8: **else**
9: **if** $\exists$ p's neighbor q with $C_{p,q} \geq u(e_{i,j})$ and tasks_allocated_to(q) $\leq M$ **then**
10: $m(t_k) \leftarrow q$
11: map_next(k,q)
12: $C_{p,q} \leftarrow C_{p,q} - u(e_{i,j})$
13: **else**
14: mapping failed
15: **end if**
16: **end if**
17: **end while**
18: **return**
19:
20: **function** map()
21: $m(t_1) \leftarrow 1$
22: map_next(1,1)
23: **return** m

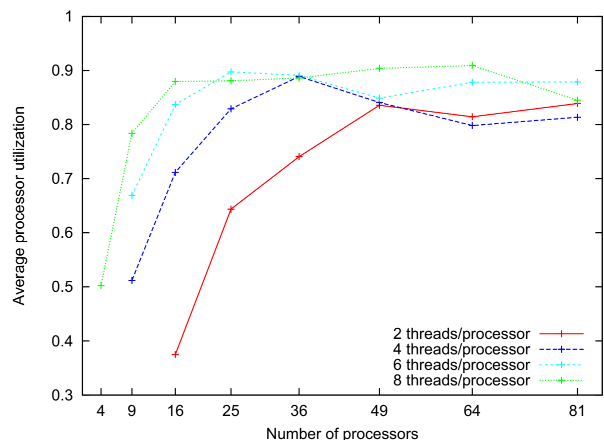


Figure 6: Processor utilization increases with number of processors indicating an effective use of system resources.

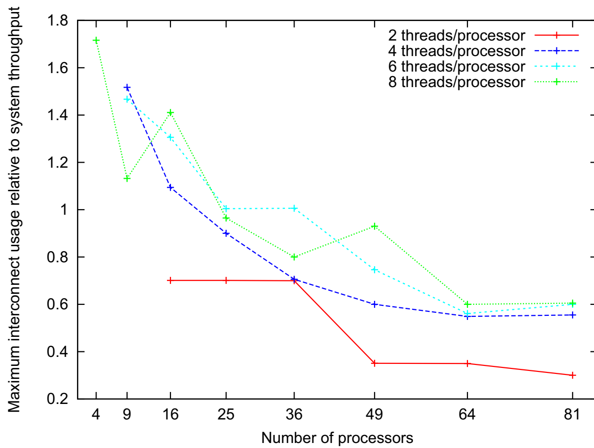


Figure 7: Inter-processor connection utilization.

ure 7. The figure shows the maximum interconnect usage relative to system throughput. One important observation form this figure is that connection utilization drops with increasing number of processors. This effect is due to higher levels of replication that lead to lower edge utilization. This results indicates the scalability of proposed packet processing platform. Even with a large number of processors, a simple local interconnect for inter-processor communication suffices.

6. SUMMARY AND CONCLUSION

Flexible packet processing is important in the current Internet as well as in future generation networks. To counter the difficulties of programming increasingly complex packet processing platforms, we propose a packet processor design that manages processing context in hardware. The resulting processor is easy to program and perform efficiently. Our design space exploration shows that we can achieve scalable systems with a large number of packet processors. The management of such a highly parallel system can be handled by a runtime system that uses dynamic task mapping. Our proposed mapping algorithm shows that processors can be utilized effectively and low interconnect speed suffice for efficient operation.

We believe that this packet processing system present an important step towards router deigns with data paths that are simpler to program and operate than in current networks.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0447873.

7. REFERENCES

[1] Anderson, T., Peterson, L., Shenker, S., and Turner, J. Overcoming the Internet impasse through virtualization. *Computer 38*, 4 (Apr. 2005), 34–41.

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

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

[4] Eatherton, W. 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).

[5] Feldmann, A. Internet clean-slate design: what and why? *SIGCOMM Computer Communication Review 37*, 3 (July 2007), 59–64.

[6] Ganapathy, S., and Wolf, T. 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.

[7] Goglin, S. D., Hooper, D., Kumar, A., and Yavatkar, R. Advanced software framework, tools, and languages for the IXP family. *Intel Technology Journal 7*, 4 (Nov. 2003), 64–76.

[8] Grunewald, M., Le, D. K., Kastens, U., Niemann, J.-C., Porrmann, M., Ruckert, U., Slowik, A., and Thies, M. Network application driven instruction set extensions for embedded processing clusters. In *Proc. of the International Conference on Parallel Computing in Electrical Engineering (PARELEC)* (Dresden, Germany, Sept. 2004), pp. 209–214.

[9] Hutchinson, N. C., and Peterson, L. L. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering 17*, 1 (Jan. 1991), 64–76.

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

[11] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The Click modular router. *ACM Transactions on Computer Systems 18*, 3 (Aug. 2000), 263–297.

[12] Kokku, R., Riché, T., Kunze, A., Mudigonda, J., Jason, J., and Vin, H. 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).

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

[14] Lockwood, J. W., McKeown, N., Watson, G., Gibb, G., Hartke, P., Naous, J., Raghuraman, R., and Luo, J. 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, June 2007), pp. 160–161.

[15] Milliken, W. C., and Dietz, J. Execution unit for a network processor. United States Patent 7,289,524, Oct. 2007.

[16] Ramaswamy, R., and Wolf, T. PacketBench: A tool for workload characterization of network processing. In *Proc. of IEEE 6th Annual Workshop on Workload Characterization (WWC-6)* (Austin, TX, Oct. 2003), pp. 42–50.

[17] Shah, N., Plishker, W., Ravindran, K., and Keutzer, K. NP-Click: A productive software development approach for network processors. *IEEE Micro 24*, 5 (Sept. 2004), 45–54.

[18] Spalink, T., Karlin, S., Peterson, L., and Gottlieb, Y. Building a robust software-based router using network processors. In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP)* (Banff, AB, Oct. 2001), pp. 216–229.

[19] Taylor, D. E., Turner, J. S., Lockwood, J. W., and Horta, E. L. Dynamic hardware plugins: Exploiting reconfigurable hardware for high-performance programmable routers. *Computer Networks 38*, 3 (Feb. 2002), 295–310.

[20] Tennenhouse, D. L., and Wetherall, D. J. Towards an active network architecture. *ACM SIGCOMM Computer Communication Review 26*, 2 (Apr. 1996), 5–18.

[21] Wolf, T. Challenges and applications for network-processor-based programmable routers. In *Proc. of IEEE Sarnoff Symposium* (Princeton, NJ, Mar. 2006).

[22] Wolf, T., Weng, N., and Tai, C.-H. Run-time support for multi-core packet processing systems. *IEEE Network 21*, 4 (July 2007), 29–37.

[23] Wu, Q., and Wolf, T. 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).

[24] Wu, Q., and Wolf, T. 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).