Network Processor: Architecture, Performance Evaluation and Applications

Yan Luo
Yan_Luo@uml.edu
http://faculty.uml.edu/yluo/

Electrical and Computer Engineering
University of Massachusetts Lowell
Outline

- Network Processor Architecture
- NePSim Simulator
- Low Power Designs
- Content-Aware Switch
Packet Processing in the Future Internet

Future Internet

- More packets
- Complex packet processing

ASIC

- High processing power
- Support wire speed
- Programmable
- Scalable
- Optimized for network applications
  - ...

General-Purpose Processors
What is Network Processor?

- Programmable processors optimized for network applications and protocol processing
  - High performance
  - Programmable & Flexible
  - Fast time-to-market

Main players: AMCC, Intel, Hifn, Ezchip, Agere

Applications of Network Processors

- DSL modem
- Wireless router
- VoIP terminal
- Printer server

Core router
Edge router
VPN gateway

Internet
## Commercial Network Processors

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Product</th>
<th>Line speed</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMCC</td>
<td>nP7510</td>
<td>OC-192/10 Gbps</td>
<td>Multi-core, customized ISA, multi-tasking</td>
</tr>
<tr>
<td>Intel</td>
<td>IXP2850</td>
<td>OC-192/10 Gbps</td>
<td>Multi-core, h/w multi-threaded, coprocessor, h/w accelerators</td>
</tr>
<tr>
<td>Hifn</td>
<td>5NP4G</td>
<td>OC-48/2.5 Gbps</td>
<td>Multi-threaded multiprocessor complex, h/w accelerators</td>
</tr>
<tr>
<td>EZchip</td>
<td>NP-2</td>
<td>OC-192/10 Gbps</td>
<td>Classification engines, traffic managers</td>
</tr>
<tr>
<td>Agere</td>
<td>PayloadPlus</td>
<td>OC-192/10 Gbps</td>
<td>Multi-threaded, on-chip traffic management</td>
</tr>
</tbody>
</table>
Typical Network Processor Architecture

SDRAM (e.g. packet buffer)  SRAM (e.g. routing table)

Network interfaces

Bus

Co-processor  H/w accelerator

Network Processor
Hardware Multithreading

Multithreading hides memory latency

Thread 0
Thread 1
Memory Access

Time

running
ready
sleeping
yield
mem req
mem done
Network Processor Research Overview

- NPs have become popular and attracted more and more attention
- Performance has been the primary interest of the NP community
- Power consumption of NPs is becoming a big concern

- A ST200 edge router can support up to 8 NP boards each of which consumes 95~150W. The total power of such a 88.4cm x 44cm x 58cm router can reach **2700W** when two chassis are supported in a single rack! – *Laurel Networks ST series router data sheet*
NP Research Tools

- Intel IXA SDK
  + accuracy, visualization
  - close-source, low speed, inflexibility, no power model

- SimpleScalar
  + open-source, popularity, power model (wattch)
  - disparity with real NP, inaccuracy

- NePSim
  + open-source, real NP, power model, accuracy
  - currently target IXP1200 only
Objectives of NePSim

- An open-source simulator for a real NP (Intel® IXP1200, later IXP2400/2800…)
- Cycle-level accuracy of performance simulation
- Flexibility for users to add new instructions and functional units
- Integrated power model to enable power dissipation simulation and optimization
- Extensibility for future NP architectures
- Fast simulation speed
NePSim Overview

Benchmark program

ME C compiler (SDK)

IXP1200  NePSim  Performance Statistics

host C compiler

NePSim source code
NePSim Software Architecture

- Microengine (six)
- Memory (SRAM/SDRAM)
- Network Device
- Debugger
- Statistic
- Verification
NePSim Internals

P0 → P1 → P2 → P3 → P4

Inst. lookup, Inst. decode, Read operand, ALU, gen mem addr, Retire, gen mem Command

Wake up sleeping threads

Event Queue

SRAM controller

SDRAM controller

arbiter

arbiter

I: instruction
C: command
E: event
Verification of NePSim

IXP1200

23990 inst. (pc=129) executed
24008 sram req issued
24009 ....

NePSim

23990 inst. (pc=129) executed
24008 sram req issued
24009 ....

Performance Statistics

Assertion Based Verification
(Linear Temporal Logic/Logic Of Constraint)

## Power Model

<table>
<thead>
<tr>
<th>H/W component</th>
<th>Model Type</th>
<th>Tool</th>
<th>Configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR per ME</td>
<td>Array</td>
<td>XCacti</td>
<td>2 64-entry files, 1 read/write port per file</td>
</tr>
<tr>
<td>XFER per ME</td>
<td>Array</td>
<td>XCacti</td>
<td>4 32-entry files, 1 read/write port per file</td>
</tr>
<tr>
<td>Control register per ME</td>
<td>Array</td>
<td>XCacti</td>
<td>1 32-entry file, 1 read/write port</td>
</tr>
<tr>
<td>Control store, scratchpad</td>
<td>Cache w/o tag path</td>
<td>XCacti</td>
<td>4KB, 4byte per block, direct mapped, 10-bit address</td>
</tr>
<tr>
<td>ALU , shifter</td>
<td>ALU and shifter</td>
<td>Wattch</td>
<td>32bit</td>
</tr>
<tr>
<td>Command FIFO, command queue in controller,</td>
<td>Array</td>
<td>Wattch</td>
<td>See paper</td>
</tr>
<tr>
<td>etc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Command bus arbiter, context arbiter</td>
<td>Matrix, rr arbiter</td>
<td>Orion</td>
<td>See paper</td>
</tr>
</tbody>
</table>
Benchmarks

- Ipfwdr
  - IPv4 forwarding (header validation, trie-based lookup)
  - Medium SRAM access

- url
  - Examining payload for URL pattern, used in content-aware routing
  - Heavy SDRAM access

- Nat
  - Network address translation
  - Medium SRAM access

- Md4
  - Message digest (compute a 128-bit message “signature”)
  - Heavy computation and SDRAM access
Performance implications

- More MEs do not necessarily bring performance gain
- More ME cause more memory contention
- ME idle time is abundant (up to 42%)
- Faster ME core results in more ME idle time with the same memory
- Non-optimal rcv/xmit configuration for NAT (transmitting ME is a bottleneck)

Throughput vs number of MEs at 232MHz

Throughput vs number of MEs at 464MHz

Idle time vs ME/memory speed ratio
Power breakdown

- Power dissipation by rcv and xmit MEs is similar across benchmarks
- Transmitting MEs consume ~5% more than receiving
- ALU consumes significant power ~45% (wattch model)
- Control store uses ~28% (accessed almost every cycle)
- GPRs burn ~13%, shifter ~7%, static ~7%
Power efficiency observations

- Power consumption increases faster than performance
- More MEs/threads bring more idle time due to memory contention

We can reduce power consumption of MEs while they waiting for memory accesses
Dynamic Voltage Scaling in NPs

- During the ME idle time, all threads are put to "wait" state and the MEs are running with the lowest activity.
- Applying DVS while MEs are not very active can reduce the total power consumption substantially.
- DVS control scheme
  - Observes the ME idle time (%) periodically.
  - When idle > threshold, scale down the voltage and frequency (VF in short) by one step unless the minimum allowable VF is hit.
  - Idle < threshold, scale up the VF by one step unless they are at maximum allowable values.
DVS Power-performance

- Initial VF=1.3V, 600MHz
- DVS period: every 15K, 20K or 30K cycles make a DVS decision to reduce or increase FV.
- Up to 17% power savings with less than 6% performance loss
- On average 8% power saving with <1% performance degradation

Power and performance reduction by DVS
Real-time Traffic Varies Greatly

- Shutdown unnecessary PEs, re-activate PEs when needed
- Clock gating retains PE instructions

Indicators of Gating/Activating PEs

- Length of thread queue
- Fullness of internal buffers
PE Shutdown Control Logic

If (thread_queue_length > T)
  increment counter;
If (counter exceeds threshold)
  { turn-off-a-PE;
    decrement threshold }
If (buffer is full)
  { turn-on-a-PE;
    increment threshold }
Challenges of Clock Gating PEs

- Terminating threads safely
  - Threads request memory resources
  - Stop unfinished threads result in resource leakage
- Reschedule packets to avoid “orphan” ports
  - Static thread-port mapping prohibits shutting down PEs
  - Dynamically assign packets to any waiting threads
- Avoid “extra” packet loss
  - Burst packet arrival can overflow internal buffer
  - Use a small extra buffer space to handle burst
Experiment Results of Clock Gating

<4% reduction on system throughput
NePSim 2.0

- Extension of NePSim to model IXP2400/2800
- ME instruction set V. 2
- Modularized Network-On-Chip (bus, crossbar etc.)
- Power modeling of SRAM/DRAM
- Graphical user interface for debugging and monitoring
Design and Implementation of A Content-aware Switch using A Network Processor

Li Zhao, Yan Luo, Laxmi Bhuyan
University of California, Riverside
Ravi Iyer
Intel Corporation
Outline

- Motivation
- Background
- Design and Implementation
- Measurement Results
- Conclusions
Content-aware Switch

- Front-end of a web cluster, one VIP
- Route packets based on layer 5 information
  - Examine application data in addition to IP & TCP
- Advantages over layer 4 switches
  - Better load balancing: distribute packets based on content type
  - Faster response: exploit cache affinity
  - Better resource utilization: partition database
Processing Elements in Content-aware Switches

- **ASIC (Application Specific Integrated Circuit)**
  - High processing capacity
  - Long time to develop
  - Lack the flexibility

- **GP (General-purpose Processor)**
  - Programmable
  - Cannot provide satisfactory performance due to overheads on interrupt, moving packets through PCI bus, ISA not optimized for networking applications

- **NP (Network Processor)**
  - Operate at the link layer of the protocol, optimized ISA for packet processing, multiprocessing and multithreading → high performance
  - Programmable so that they can achieve flexibility
Outline

- Motivation
- Background
  - NP architecture
  - Mechanism to build a content-aware switch
- Design and Implementation
- Measurement Results
- Conclusion
Background on NP

Hardware
- Control processor (CP): embedded general purpose processor, maintain control information
- Data processors (DPs): tuned specifically for packet processing
- Communicate through shared DRAM

NP operation on packets
- Packet arrives in receive buffer
- Header Processing
- Transfer the packet to transmit buffer
Mechanisms to Build a CA Switch

- TCP gateway
  - An application level proxy
  - Setup 1\textsuperscript{st} connection w/ client, parses request $\rightarrow$ server, setup 2\textsuperscript{nd} connection w/ server
  - Copy overhead

- TCP splicing
  - Reduce the copy overhead
  - Forward packet at network level between the network interface driver and the TCP/IP stack
  - Two connections are spliced together
  - Modify fields in IP and TCP header
Operations on a Content-Aware Switch
Outline

- Motivation
- Background
- Design and Implementation
  - Discussion on design options
  - Resource allocation
  - Processing on MEs
- Measurement Results
- Conclusion
Design Options

- **Option 0:** GP-based (Linux-based) switch
- **Option 1:** CP setup & and splices connections, DPs process packets sent after splicing
  - Connection setup & splicing is more complex than data forwarding
  - Packets before splicing need to be passed through DRAM queues
- **Option 2:** DPs handle connection setup, splicing & forwarding
IXP 2400 Block Diagram

- XScale core
- Microengines (MEs)
  - 2 clusters of 4 microengines each
- Each ME
  - run up to 8 threads
  - 16KB instruction store
  - Local memory
- Scratchpad memory, SRAM & DRAM controllers
Resource Allocation

- Client-side control block list: record states for connections between clients and switch, states for forwarding data packets after splicing.
- Server-side control block list: record state for connections between server and switch.
- URL table: select a back-end server for an incoming request.
Processing on MEs

- Control packets
  - SYN
  - HTTP request
- Data packets
  - Response
  - ACK
Outline

- Motivation
- Background
- Design and Implementation
- Measurement Results
- Conclusion
Experimental Setup

- Radisys ENP2611 containing an IXP2400
  - XScale & ME: 600MHz
  - 8MB SRAM and 128MB DRAM
  - Three 1Gbps Ethernet ports: 1 for Client port and 2 for Server ports
- Server: Apache web server on an Intel 3.0GHz Xeon processor
- Client: Httperf on a 2.5GHz Intel P4 processor
- Linux-based switch
  - Loadable kernel module
  - 2.5GHz P4, two 1Gbps Ethernet NICs
Measurement Results

- Latency reduced significantly
  - 83.3% (0.6ms $\rightarrow$ 0.1ms) @ 1KB
- The larger the file size, the higher the reduction
  - 89.5% @ 1MB file
## Analysis – Three Factors

<table>
<thead>
<tr>
<th>Linux-based</th>
<th>NP-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt: NIC raises an interrupt once a packet comes</td>
<td>polling</td>
</tr>
<tr>
<td>NIC-to-mem copy</td>
<td>No copy: Packets are processed inside w/o two copies</td>
</tr>
<tr>
<td>Xeon 3.0Ghz Dual processor w/ 1Gbps Intel Pro 1000 (88544GC) NIC, 3 us to copy a 64-byte packet by DMA</td>
<td></td>
</tr>
<tr>
<td>Linux processing: OS overheads</td>
<td>IXP processing: Optimized ISA</td>
</tr>
<tr>
<td>Processing a data packet in splicing state: 13.6 us</td>
<td>6.5 us</td>
</tr>
</tbody>
</table>
Measurement Results

- Throughput is increased significantly
  - 5.7x for small file size @ 1KB, 2.2x for large file @ 1MB
- Higher improvement for small files
  - Latency reduction for control packets > data packets
  - Control packets take a larger portion for small files
An Alternative Implementation

- **SRAM**: control blocks, hash tables, locks
  - Can become a bottleneck when thousands of connections are processed simultaneously; Not possible to maintain a large number due to its size limitation

- **DRAM**: control blocks, SRAM: hash table and locks
  - Memory accesses can be distributed more evenly to SRAM and DRAM, their access can be pipelined; increase the # of control blocks that can be supported
Measurement Results

- Fix request file size @ 64 KB, increase the request rate
- 665.6Mbps vs. 720.9Mbps
Conclusions

- Designed and implemented a content-aware switch using IXP2400
- Analyzed various tradeoffs in implementation and compared its performance with a Linux-based switch
- Measurement results show that NP-based switch can improve the performance significantly
Backups
TCP Splicing

client

SYN(CSEQ)

DATA(CSEQ+1)

ACK(CSEQ+1)

ACK(DSEQ+1)

step1

step2

step3

step4

step5

step6

step7

step8

content switch

SYN(DSEQ)

ACK(CSEQ+1)

server

SYN(CSEQ)

DATA(CSEQ+1)

ACK(SSEQ+1)

DATA(SSEQ+1)

ACK(CSEQ+lenR+1)

ACK(SSEQ+lenD+1)

ACK(DSEQ+lenD+1)

lenR: size of http request.
lenD: size of return document
TCP Handoff

- Migrate the created TCP connection from the switch to the back-end server
  - Create a TCP connection at the back-end without going through the TCP three-way handshake
  - Retrieve the state of an established connection and destroy the connection without going through the normal message handshake required to close a TCP connection
- Once the connection is handed off to the back-end server, the switch must forward packets from the client to the appropriate back-end server

12/18/05 Yan Luo, ECE of UMass Lowell
NePSim Overview

- ME C program
- Compiler-generated microcode
- Microcode program
- Microcode assembler
- Parser
- Internal format
- NePSim
- Stats Results
- host C compiler
- NePSim source code
- ME C compiler SDK
NePSim Internals (I)

- **Instruction**
  - Opcode: ALU, memory ref., CSR access etc.
  - Operands: GPR, XFER, immed,
  - Shift: shift amount
  - Optional token: ctx_swap, ind_ref, …

- **Command (for memory, fbi accesses)**
  - Opcode: sram_read, sram_write, sdram_read, …
  - Thread id: ME, thread
  - Functional unit: sram, sdram, scratchpad, fbi
  - Address: source or destination address
  - Reg: source or destination XFER register
  - Optional token: ctx_swap, ind_ref, …

- **Event**
  - <cycle time, command>