A Preliminary Performance Model for Optimizing Software Packet Processing Pipelines

Ankit Bhardwaj, Atul Shree, Bhargav Reddy V and Sorav Bansal
Indian Institute of Technology Delhi

APSys 2017
Software based Packet Processing

• Increasing popularity of **Software Defined Networks (SDN)**.
  ○ Flexibility
  ○ Develop and test new functionality
  ○ Use of commodity hardware

• Programming Environment
  ○ Imperative languages, ex: C
  ○ Stream programming languages, ex: DSLs like P4 and Click
Problem Statement

Can we bridge this gap with the help of a compiler?

- Manual optimizations are time consuming and repetitive
- Rapid changes in the architecture makes the problem harder
Motivation

- A compiler needs a performance model of underlying system to make optimizations and code transformations.
- Focus is on scheduling and prefetching related optimizations.
- Apply optimizations for single CPU core
  - Multi-queue support in NIC enables linear scalability

Up to 57% performance gain with these optimizations.
Background

- **P4 (Programming Protocol Independent Packet Processing)**
  - A DSL for software packet processing
  - Active community
  - Adoption is growing swiftly in industry and academia

- **P4C**
  - A prototype compiler for P4 which generates DPDK based C code
  - One of the early compiler for P4

- **Intel DPDK**
  - A set of libraries and drivers for fast packet processing
Example in P4

header_type ethernet_t {
fields {
    dstAddr : 48;
    srcAddr : 48;
    etherType : 16;
}
}

table dmac {
    reads {
        ethernet.dstAddr : exact;
    }
    actions {forward; bcast;}
    size : 512;
}

action forward(port) {
    modify_field(standard_metadata.egress_port, port);
}

action bcast() {
    modify_field(standard_metadata.egress_port, 100);
}

Input -> Parser -> Table -> Match -> Action -> Output
Compilation Phases

P4 Program → HLIR → Intermediate Rep. → P4C → Core code using HAL calls → GCC → Switch

- Provided by P4 developers
- Hardware Abstraction Library in DPDK
- Core code using HAL calls
- Pipeline related optimizations
- Standard GCC optimizations
Superscalar out-of-order execution

Schedule of memory accesses

Re-Order Buffer

MEM

CPU

A Serviced

B Serviced

CPU Stalled

CPU Stalled
Superscalar out-of-order execution

Schedule of memory accesses

Re-Order Buffer

MEM

CPU

A Serviced
B Serviced

CPU active at all times
Packet Processing Pipeline
Packet Processing Pipeline
Packet Processing Pipeline

![Diagram of Packet Processing Pipeline](image-url)
Packet Processing Pipeline
Packet Processing Pipeline
Packet Processing Pipeline
Packet Processing Pipeline
Packet Processing Pipeline
Packet Processing Pipeline
Packet Processing Pipeline
Packet Processing Pipeline
Packet Processing Pipeline
Packet Processing Pipeline
Packet Processing Pipeline

1. Exploit **IO Parallelism** between **NIC** and **Main Memory**
2. Exploit **Memory Parallelism** between **CPU** and **Memory**
Packet Processing Pipeline

1. Improvement due to IO parallelism 20% - 57%
2. Additional improvement of 21% - 23% due to memory parallelism
NIC-Memory I/O Parallelism

NIC

RX Queue / MEM

Input

To Processing

Batching
NIC-Memory I/O Parallelism

Input → NIC → RX Queue / MEM → To Processing

Batching
NIC-Memory I/O Parallelism

NIC

RX Queue / MEM

Input

To Processing

Batching
NIC-Memory I/O Parallelism

Input

NIC

RX Queue / MEM

To Processing

Batching
NIC-Memory I/O Parallelism

- **Input**
- **NIC**
- **RX Queue / MEM**
- **To Processing**

**Batching**
Batching($B$) == Loop Fission

```
sub app {
    for( i=0; i<B; i++){
        p = read_from_input_NIC();
        p = process_packet(p);
        write_to_output_NIC(p);
    }
}
```
Batching($B$) == Loop Fission

IO, CPU and Memory work in parallel
Memory Level Parallelism

From RX Queue

MEM

CPU

To TX Queue
Memory Level Parallelism

From RX Queue

MEM

CPU

To TX Queue
Memory Level Parallelism

From RX Queue

MEM

CPU

To TX Queue
Memory Level Parallelism

From RX Queue

MEM

CPU

To TX Queue
Memory Level Parallelism

From RX Queue

MEM

CPU

To TX Queue
Memory Level Parallelism

From RX Queue

MEM

CPU

To TX Queue
Memory Level Parallelism
Memory Level Parallelism

From RX Queue

MEM

CPU

To TX Queue
Memory Level Parallelism

From RX Queue → MEM → CPU ← → To TX Queue
Memory Level Parallelism

From RX Queue

MEM

CPU

To TX Queue
Memory Level Parallelism
Memory Level Parallelism
Memory Level Parallelism

From RX Queue

MEM

CPU

To TX Queue
Loop Fission for Sub-Batching($b$)

```
sub process_packet(p) {
    for (i=0; i<B; i++) {
        t1 = lookup_table1(p[i]);
        t2 = lookup_table2(p[i], t1);
        ...
    }
}
```
sub process_packet(p) {
    for( i=0; i<B; i++){
        t1 = lookup_table1(p[i]);
        t2 = lookup_table2(p[i],t1);
        ...
    }
}

sub process_packet(p) {
    for( i=0; i<B; i+=b){
        for( j=i; j<i+b; j++)
            t1[j-i] = lookup_table1(p[j]);
        for( j=i; j<i+b; j++)
            t2=lookup_table2(p[j],t1[j-i]);
        ...
    }
}
Reduce Stall Time

Memory Stall

CPU

Demand Access

Request Served

Memory
Reduce Stall Time

Memory Stall

CPU

Demand Access

Request Served

Memory

Issue Prefetch

Demand Access

Request Served

CPU

Memory

Prefetching
Reduce Stall Time with Prefetching

- Fixed Prefetch Distance irrespective of Application Nature

```c
for( i=0; i<B; i++){
    key_hash[i] = hash_compute(key[i]);
    prefetch(bucket(key-hash[i]));
}

for( i=0; i<B; i++){
    val[j] = hash_lookup(key_hash[j]);
}
```
Reduce Stall Time with Prefetching

- Sub-batch size allows flexible Prefetch distance

```
for( i=0; i<B; i++){
    key_hash[i] = hash_compute(key[i]);
    prefetch(bucket(key-hash[i]));
}

for( i=0; i<B; i++){
    val[j] = hash_lookup(key_hash[j]);
}
```

```
for( i=0; i<B; i+=b){
    for( j=i; j<i+b; j++){
        key_hash[j] = hash_compute(key[j]);
        prefetch(bucket(key_hash[j]));
    }

    for( j=1; j<i+b; j++){
        val[j] = hash_lookup(key_hash[j]);
    }
}
```
Impact of Prefetch Distance

Early Prefetch | Ideal Prefetch | Late Prefetch | Demand Access

Memory Access Time

Cache Contention | Memory Stall
Evaluation
Setup

Hardware (Client and Server)

- **8 cores**, each works at **2.6 GHz**
- **32Kb L1**, **256Kb L2**, & **20Mb L3** cache

![Diagram of hardware components]
## Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>#Entries</th>
<th>#Lookups</th>
<th>Boundedness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer 2 Forwarding</td>
<td>16 M</td>
<td>2</td>
<td>Memory Bound</td>
</tr>
<tr>
<td>Named Data Networking</td>
<td>10 M</td>
<td>1-4</td>
<td>Memory Bound</td>
</tr>
<tr>
<td>IPv4 Forwarding</td>
<td>528 K</td>
<td>1-2</td>
<td>L3 Bound</td>
</tr>
<tr>
<td>IPv6 Forwarding</td>
<td>200 K</td>
<td>4-6</td>
<td>L3 Bound</td>
</tr>
<tr>
<td>L2 Forward encryption/decryption</td>
<td>4</td>
<td>1</td>
<td>CPU Bound</td>
</tr>
</tbody>
</table>
Experiments and Results
Effect of Batching and Prefetching
Sensitivity of Throughput to $B$

![Graph showing the sensitivity of throughput to batch size in the L2Fwd Application. The graph indicates that throughput increases with increasing batch size up to a certain point, after which it starts to decrease. The x-axis represents batch size in multiples of 8, ranging from 8 to 256, and the y-axis represents throughput in Mpps (millions of packets per second), ranging from 12.50 to 17.00.]

- IO Parallelism
- Cache pressure
Prefetching Performance

![Graph showing throughput vs. batch size for different application types: IO Bound (dotted line) and Memory Bound (solid line). The graph indicates that the throughput increases with batch size up to a certain point and then decreases, with the optimal batch size being around 32 for both types. The labels indicate that $b = 1$ and $b = B$.](image-url)
Sensitivity of Throughput to $b$

**B = 128**

Optimal Sub-batch Size

L2Fwd Application
Comparison with other related work

- ★ 50% better than vanilla-P4C
- ★ 15%-59% better than G-Opt
- ★ Equal or better than Hand Optimized Code
What is the Performance Model?
Nature of Applications

- CPU Bound
- IO Bound
- Memory Bound

Typical Application
Performance Model

- Can be viewed as a **queuing system** with three components
- Let the service rate of CPU, CPU-Memory, I/O-Memory DMA interface be \( c, m, d \).
- Assume that components can work independently
- Throughput = \( \min(c, m^b, d^B) \)
- Based on the nature of application, **predict** \( b \) & \( B \) and generate the optimized code.
Conclusion

- **Scheduling** and **prefetching** optimizations
- Predict $b$ & $B$ based on application nature
- Significant Performance improvement over vanilla P4C and other previous work.
- Current Model is based on coarse grained experiments
Conclusion and Future Work

- **Scheduling** and **prefetching** optimizations
- Predict $b$ & $B$ based on application nature
- Significant Performance improvement over Vanilla P4C and other previous work.
- Current Model is based on coarse grained experiments
  - Perform fine grained experiments to get low level understanding about IO, Memory and CPU
  - Explore optimizations other than scheduling and prefetching, and their interplay
Thank You