Performance model for mapping processing tasks to OpenFlow switch resources

Omar El Ferkouss*, Rachia Ben Ali†, Yves Lemieux†, Cherkaoui Omar*
*Université du Québec à Montréal (UQAM)
†Ericsson Research Canada

Abstract—In a recent effort to push forward the powerful concept of software defined networks, OpenFlow has gained a lot of popularity as a practical approach to split the data and the control planes by standardizing an open interface that allow remote software controllers to dictate the forwarding behavior of network devices. The latest flexible version 1.1 of OpenFlow is limited to software forwarding plane implementations. In order to deliver high performance, we implement an OpenFlow v1.1 hardware forwarding plane based on network processors. Delivering the optimal performance requires finding the optimal mapping of OpenFlow tasks to hardware resources which is already known as an NP-hard combinatorial problem. In our work, we propose a performance model that helps choose a better mapping without the burden of implementing and comparing all possible mappings on network processor. Our model assumes that the performance bottleneck of the hardware forwarding pipeline comes from the lookup tasks. This is generally the case of OpenFlow lookup tasks based on more than 14 tuple headers and requiring high latency external memories in order to provide a large number of possible flow entries. Our model, validated using real hardware implementations comparisons, shows that the lookup tasks that use the same external memories in the same pipeline are not worth parallelizing. In fact, mapping them to different parallel processing elements will only increase the response time of the lookup memory which will slow down the forwarding pipeline.

I. INTRODUCTION

Applications designers aims at achieving good performance over a network node. Indeed, each network node is composed of a set of available resources (Processing elements, memory, Bus, ...). An application is a set of tasks that are executed on a pipeline or parallel fashion. The usage of the node resources by the application tasks is really critical and could affect the performance of the node. Thus, the mapping of the tasks to the node resources should be thought of. However, the mapping is not unique and therefore we have to find the best mapping that ensures better external performance and better usage of the node resources. Finding the bottleneck helps to find the best mapping that ensures the best performance.

In any network forwarding system, the performance bottleneck can be one of the three following types: an egress bandwidth, a packet processing or a lookup memory bottlenecks. If we attempt to remove the bottleneck in one network area we may just move it somewhere-else. In general, bandwidth bottlenecks can be resolved by using high speed links, aggregating physical links using port channels, or even using multiple paths to combine the bandwidth of several paths. Processing bottlenecks can also be resolved using massive parallelism and deep pipelining by adding more specialized PEs (processing elements) in the system. We usually need to use relatively high access and low bandwidth off-chip memories.

This paper proposes a performance model based on this memory management assumption in order to provide a heuristic tool that helps the network node designer finding the best memory mapping. More particularly, it will helps him to find out which shared memory table lookup task to PE mapping will provide a better performance without taking into account all the possible mapping combinations. In fact, the general mapping problem of processing tasks to PE is already shown as an NP-Hard problem.

The proposed model is evaluated by both simulation and the use of real hardware. The evaluation that is made is for an implementation of the version 1.1 of OpenFlow [5], [7] over a pizza box containing a network processor. The proposed performance model tries to find how to map the pipeline of lookups in the OpenFlow 1.1 multiple tables with the network processor resources.

The remainder of this paper is divided as follow: related existing works that treat the same problematic is presented in section II followed by the proposed performance model for memory management in section III. The experimentation results obtained by simulation and with a real forwarding node are presented in section IV. Finally, the paper concludes in section V with our proposed model for optimized memory management while considering both best usage of resources and improved packet processing performance.

II. RELATED WORKS

In [16], a performance model is already presented for a hybrid topology based system. This model is intended for standard network applications. In our case, however, we will try to adapt it so that it considers much more general processing tasks in generic applications and their flexible mapping to the hardware resources of a general packet processing system.

In [17], a simple performance model is proposed for general-purpose multi-core based systems. The simplicity of the model aims to quickly provide performance metrics computations. These metrics will serve to adapt the workload-to-
processor mappings during the runtime depending on workload dynamics.

In [14] authors propose an heuristic algorithm to map a processing task for a system with heterogeneous resources. This algorithm takes into consideration the communication between the resources (processors for instance) and the parallelism in the execution of tasks. It takes as input a task graph showing the pipeline of tasks and a resource graph that models the resources and calculate the optimal mapping of tasks with the available resources. Our approach however, is more focused in how to map the processing tasks to the processing elements in order to resolve the lookup task bottleneck. More heuristics models for distributed systems are also proposed in [6], [12], [13].

In other instances, the resource allocation to different tasks may also be treated for QoS purposes. This is the case with [4] that proposes an algorithm for the mapping of tasks to a multiprocessor based on each task QoS. Each task requires a QoS level and the algorithm tries for that purpose, to maximize the overall system QoS. Similarly, a Q-RAM (QoS based Resource Allocation Model) is proposed in [10]. It aims at allocating the available resources. Our approach however, is more focused more specifically on NP-based packet processing systems.

We consider NPs with hybrid topologies in order to generalize the pipeline of tasks and a resource graph that models the communication mechanism that are based on shared memory.

Furthermore, we assume that following a directed acyclic graph of execution described later, each network application A is decomposed in a sequence of instructions clustered in blocks of processing tasks that can be described as $\tau_l(A), l = 1..L$. In this clustering, most of the process intensive instruction sequences as well as the data dependencies are confined into the same block thus profiting from pipeline and limiting the communication overhead between blocks that are running on different PEs.

In the execution directed acyclic graph of application A, denoted $G(A)$, if a processing task $\tau_l(A)$ depends on or requires the previous execution of $\tau_{l-1}(A)$, then $(\tau_{l-1}(A), \tau_l(A)) \in E(G(A))$, i.e. is a directed edge of $G(A)$.

We denote $l(\tau_l(A), PE_{i,j}) \in Map$ the processing time of a processing task $\tau_l(A)$ of a network application A when the $\tau_l(A)$ is mapped and executed by $PE_{i,j}$ following the mapping Map. This processing time is equal to the number of instructions cycles required for this processing task and denoted $C(\tau_l(A), PE_{i,j}) \in Map$.

On the other hand, the memory response time for a number $r_m(\tau_l(A))$ of reads and a number $w_m(\tau_l(A))$ of writes to a memory type $m$ (on-chip cache, off-chip SRAM, off-chip DRAM, on-chip TCAM, off-chip TCAM) is modeled as a Machine Repairman Model (MRM) [11].

MRM is a queuing network model where:

- The job servers generating the failed machines represent the different active $PE$s that generate the memory access requests.

---

### III. PERFORMANCE MODEL FOR MAPPING PROCESSING TASKS TO HARDWARE RESOURCES

#### A. System model

In this section, we provide our Memory Management Model proposal. Packet processing systems can either be NP-based systems or a general-purpose-multi-core systems. Although we propose a general model that applies to both types, we will focus more specifically on NP-based packet processing systems. We consider NPs with hybrid topologies in order to generalize to all existent architectures [15]. This hybrid topology is a generalization of both pipelined heterogeneous architectures such as Agere’s Fast Pattern Processors and parallel homogenous multiprocessors architectures such as Intel’s IXP2400. Cisco Toaster is an example of hybrid topology architectures that uses parallel pipelines of PEs. Moreover, EZchip NPs are a typical example of hybrid topology architectures that uses a pipeline of parallel Task Optimized Processing (TOPs) elements. The latter will be used in our testbed implementation and evaluation.

For our proposed model, we assume a hybrid packet processing system with $I_j$ parallel identical PEs, i.e. running the same type of processing tasks, for each pipelining stage $j$ among the $D$ heterogeneous pipelining stages, i.e. running different types of processing tasks. This system is illustrated in Figure 1.

Each processing task belongs to a single type among $T \leq D$ defined types such as packet parsing, packet classification, packet lookup, packet modification types, ... etc.

In general, network applications are profiled and decomposed to processing tasks (a set of clustered packet-processing instructions) so that each one of them is profiled to belong to a specific type among the vertical $T$ processing task types and then mapped to one of the horizontal $D$ pipeline stages.

---

![Fig. 1. Pipeline of parallel PEs in a hybrid packet processing system](image-url)
• The repairman servers that repairs the failed machines represent the memory channels serving the memory access requests as illustrated in Figure 2.

MRM is also a closed queuing network with finite population of memory requesters, since the number of memory access requests are limited by the finite processing tasks running on up to \( WD \) active PEs.

We assume that all the \( M \) memories are shared by all the \( WD \) PEs. We consider \( \sum_{j=1}^{D} I_j \) PEs (machines) subject to accessing these memories as illustrated in Figure 2. Let there be at most \( N_m \) simultaneous accesses to memory \( m \) (limited by the memory bandwidth). Note that if a specific pipeline stage \( j \) uses a type of PE that does not implement access to a specific memory \( m \), then \( I_j = 0 \). This is the case with the lookup memories such as TCAM or SRAM that are only used by specific lookup optimized PEs (the type of lookup processing tasks).

The successive access durations are assumed to be i.i.d (independent and identically distributed ) with mean time \( 1/S_m \). We assume that the memory access strategy is First Come First Serve (FCFS).

where \( L_s \) is the bottleneck latency or the latency of the pipeline stage causing the performance bottleneck. At steady state we always have \( I_j \leq I_s \) for any stage \( j \).

The total number of processing tasks (of all admitted applications) that are mapped to \( PE_{i,j} \) following a given mapping of processing tasks to PEs denoted \( Map \) is:

\[
\tau_{i,j}^{Map} = \text{card}(\tau_{i,j}^{Map})
\]

where \( \tau_{i,j}^{Map} \) is given by:

\[
\tau_{i,j}^{Map} = \{ \tau_l(A), A \in \text{Apps}, (\tau_l(A), PE_{i,j}) \in Map \}
\]

Task \( \tau \) is initially profiled to belong to a specific type of processing tasks. Therefore if a single stage is optimized of this type of processing it will be directly mapped to that stage. If more than one stage is optimized for the same type of processing, different combinations are possible in mapping several processing tasks to these stages. In general, depending on a predefined input classification and input scheduling (e.g. round robin on input ports and priority queuing for class of services defined on MPLS tag, VLAN id, etc.), the task \( \tau \) is mapped to \( \{PE_{i,j}\} \) where \( Map_{i,j} = 1 \).

Furthermore, we assume that each task \( \tau \) has its packet traffic \( \lambda_{\tau} \) that triggers its execution. In the non-saturated case, the packet service time by task \( \tau \), denoted \( L_{\tau} \), over the \( W \) parallel PEs is less than the inter packet arrival time. Thus, \( L_{\tau}/W \leq 1/\lambda_{\tau} \).

The mean service time at the slowest stage is:

\[
L_s = \frac{\sum_{\tau, Map_{i,j}=1}^{\tau_{i,j}^{Map}} \lambda_{\tau} L_{\tau}}{\sum_{\tau, Map_{i,j}=1}^{\tau_{i,j}^{Map}} \lambda_{\tau}}
\]

In order to process \( \lambda = \sum_{\tau, Map_{i,j}=1}^{\tau_{i,j}^{Map}} \lambda_{\tau} \) in a non-saturated condition, we need to have at least \( I_{s}/I_s \) parallel active PEs at the slowest stage \( s \). Where \( 1/\lambda = L_s/I_s \).

Thus:

\[
I_s = \lambda L_s = \sum_{\tau, Map_{i,j}=1}^{\tau_{i,j}^{Map}} \lambda_{\tau} L_{\tau}
\]

And for any stage \( j \), the number of parallel active PEs is:

\[
I_j = \frac{I_j L_j}{L_s}
\]

The mean access rate to memory \( m \) by processing task \( \tau \) is equal to:

\[
\sigma_{\tau,m} = \lambda_{\tau} (r_{\tau,m} + w_{\tau,m})
\]

Where \( \lambda_{\tau} \) is the average incoming packet rate processed by task \( \tau \). Where \( r_{\tau,m} \) and \( w_{\tau,m} \) are the number of reads and writes to memory \( m \) by processing task \( \tau \).

The mean access rate to memory \( m \) by processing tasks mapped to run on \( PE_{i,j} \) is equal to:

\[
\sigma_{i,j,m} = \sum_{\tau, Map_{i,j}=1}^{\tau_{i,j}^{Map}} \sigma_{\tau,m}
\]

![Fig. 2. Performance model for response time of memory of type m](image_url)

We assume that traffic flow inter-arrival time to the system is denoted \( \Delta \). During \( \Delta \) the number of active PEs, i.e. "in use", does not increase, i.e. it remains constant or decreases slightly due to flow departures. Only one flow arrival occurs at the beginning of the time interval of duration \( \Delta \). The maximal number of active PEs is constant during \( \Delta \) for each stage and denoted \( I_j \) for stage \( j \) of the pipeline.

The throughput \( S \) of the system is limited by the pipeline stage \( j = s \) causing the bottleneck, i.e. that has the minimal throughput, i.e. the maximal delay, when using the maximal number \( I_s \leq W \) of parallel PEs at stage \( s \) compared to other stages. Thus, the throughput \( S \) of the system is:

\[
S = \min(I_j/L_j) = I_s/L_s
\]
Recall that the latency of processing task $\tau$ mapped to run on a $PE_{i,j}$ includes processing latency and memory latency and is equal to:

$$L_\tau = L^{proc}_{i,j} + L^{mem}_{i,j}$$

(9)

The PE processing latency of $PE_{i,j}$ is equal to $L^{proc}_{i,j} = C_\tau$ if $Map^{*}_{i,j} = 1$.

The memory latency of $PE_{i,j}$ is equal to:

$$L^{mem}_{i,j} = \sum_{m=1..M} ((r_{T,m} + w_{T,m})R_m)$$

(10)

if $Map^{*}_{i,j} = 1$.

We assume negative exponential service times for memory access requests of type $m$ and constant service time of the $m$-type memory access time. Then, we can use the analysis in [11] to formulate the latency in accessing memory of type $m$. However the model in [11] assumes homogeneous $PES$ with the same mean access rate to memories for all $PES$. Averaging the memory access rate for all $PES$ does not provide a way to differentiate the memory response time of different mappings. Therefore we assume different $\sigma_{i,j,m}$ for different $PES$.

The model in Figure 2 is resolved by establishing the steady state equilibrium equations of state probabilities of the correspondent Markov chain. We extend the following recursive form of steady state equations to multiple memories and parameterized number of active $PES$ at each pipeline stage from [11]:

$$R_{m, I_1, I_2, ..., I_D} = (1 + Q_{m, I_1, I_2, ..., I_{D-1}})S_m$$

(11)

$$X_{m, I_1, I_2, ..., I_D} = \sum_{j=1..D} \sum_{I_j=1..I_D} \frac{1}{\sigma_{i,j,m}} R_{m, I_1, I_2, ..., I_{D-1}, I_j}$$

(12)

$$Q_{m, I_1, I_2, ..., I_D} = \frac{X_{m, I_1, I_2, ..., I_{D-1}, I_D - R_{m, I_1, I_2, ..., I_{D-1}, I_D}}}{N_m}$$

(13)

Where $S_m$ is the service time of memory $m$ and $R_{m, I}$ is its response time when there are $I_j = I[j]$ active $PES$ at stage $j$. $\frac{1}{\sigma_{i,j,m}}$ is the total number of memory access requests to memory $m$ if there are $I_j = I[j]$ active $PES$ at stage $j$. $X_{m, I}$ is the throughput of $m$-type memory requests when there are $I_j = I[j]$ active $PES$ at stage $j$.

**IV. EXPERIMENTATION RESULTS**

The evaluation of the proposed performance model is done over a pizza box containing an EZchip [3] network processor for an implementation of the version 1.1 of OpenFlow. It was already used at a GENI conference [8] and for previous other works [2], [9].

The pizza box used in the experimentation contains an EZchip NP-4 network processor. It is a pipelined architecture of network processor capable of processing 100 Gbps full rate. It also consists of task optimized processor or TOPs, each one is dedicated to process a special task. EZchip NP-4 contains four types of TOPs:

- TOPparse to parse and classify the packet.
- TOPresolve to perform lookups in the corresponding search structures.
- TOPsearch to perform lookups in the corresponding search structures.
- TOPmodify to apply the modifications to the packet.

In order to evaluate our proposed performance model, we profile the different functions of an MPLS applications and we cluster them into multiple lookup processing tasks following the multiple tables introduced in OpenFlow version 1.1. The directed acyclic graph of the multiple tables lookups referring to this MPLS applications is illustrated in Figure 3. Following this, we apply our performance model to the top path in this acyclic graph representing the Push MPLS label function. This function uses three tables (table 0, table 1 and table 2) in our design: one lookup in the TCAM (table 0) and two subsequent lookups (table 1 and 2) in the Hash memory. We then choose to compare two different types of mappings (of these lookup processing tasks) to TOPs search I and TOP search II PEs.

- The first mapping, denoted mapping 0, is a pipeline table lookup in the shared Hash memory that is done by mapping the two corresponding lookups to two different stages of the pipeline (Search I and Search II).
- Whereas in the second mapping, denoted mapping 1, we serialize tables lookup searches in Hash by mapping the two Hash lookups to the same stage of the pipeline (TOP search 1).

These two mappings are illustrated in Figure 4. Table I describe the parameter settings for the mapping performance model such as the task processing time, number of pipeline stages and parallel PEs, the system offload, ...etc. Recall that the tasks processing time are drawn from an implementation over an EZchip NP-4 (for example: TOPparse task time, lookup time, ...).

A. Numerical results comparison of two types of mappings

Numerical results in Figure 5 show that up to a certain load of 0.04 Gpps the bottleneck is the processing since the maximal latency at the bottleneck stage is constant regardless of the offered load. And then starting from an offered load of 0.05 Gpps, the bottleneck becomes the memory lookup since the latency at the bottleneck stage increases with the offered load. We also notice that mapping 1 provides a much lower bottleneck latency than mapping 0 and therefore a much higher throughput performance as illustrated in Figure 6.

B. Performance evaluation on real Hardware

Table II describes the performance results of the performance model for the two mapping:

- Mapping 0: the lookup is distributed between the TOPs.
- Mapping 1: all lookups are made in the same TOPsearch engine.

It is important to remember that that TOPsearch is the only TOP that can perform lookups in EZchip NP-4.

In this experiment, the pizza box is was connected to the IXIA [1] test equipment using 10 Gbps links and generating
traffic accordingly. We calculated the processing limit in packets/second that the pizza box can process. This limit is found by generating a traffic and increasing the traffic rate until it reaches the packet processing limit at which point the switch starts to lose packets. As shown in Table II, the mapping 0 starts to lose packets for a traffic rate of 53.6 Million packets/second which corresponds to 85 Gbps for 200 Bytes packet size while the mapping 1 starts to lose packets for a traffic rate greater than 59.5 Million packets/second. Thus, it is better to have all lookups in the same TOPSearch rather than distribute the lookups between the TOPs and therefore, the mapping 1 is better than the mapping 0 in terms of performance if we consider the MPLS protocol as application.

To sum up these results, we deduce from our performance model that serializing shared memory lookups into the same PEs using mapping 1 will incur less parallel load to this shared memory and therefore higher performance. However, this mapping 1 comes with a less pipelining effect since we use less pipeline stages in the lookup.

V. CONCLUSION

Designing applications over network nodes is a critical task and it’s not easy to find the right mapping of network

---

**Table I**

PARAMETER SETTINGS FOR THE MAPPING PERFORMANCE MODEL

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(W, D)$</td>
<td>$(8, 5)$</td>
</tr>
<tr>
<td>$L^\text{proc}_{\text{ToP parse}}$</td>
<td>30 CC</td>
</tr>
<tr>
<td>$L^\text{proc}_{\text{ToP search I}}$</td>
<td>11 CC</td>
</tr>
<tr>
<td>$L^\text{proc}_{\text{ToP resolve}}$</td>
<td>7 CC</td>
</tr>
<tr>
<td>$L^\text{proc}_{\text{ToP search II}}$</td>
<td>2 CC</td>
</tr>
<tr>
<td>$L^\text{proc}_{\text{ToP modify}}$</td>
<td>40 CC</td>
</tr>
<tr>
<td>$t_{\text{lookupTab0,T CAM}}$</td>
<td>1 ns</td>
</tr>
<tr>
<td>$t_{\text{lookupTab1,Hash}}$</td>
<td>1 ns</td>
</tr>
<tr>
<td>$t_{\text{lookupTab2,Hash}}$</td>
<td>1 ns</td>
</tr>
</tbody>
</table>

**Table II**

PROCESSING LIMIT OF THE IMPLEMENTATION OF OPENFLOW SWITCH IN MILLION PACE/SEC FOR THE TWO MAPPINGS

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping 0</td>
<td>53.6</td>
</tr>
<tr>
<td>Mapping 1</td>
<td>59.5</td>
</tr>
</tbody>
</table>

---

Fig. 3. Example of memory accesses in the processing tasks for MPLS applications

Fig. 4. Example of two mappings of shared memory lookup searches to PEs

Fig. 5. Latency of the bottleneck stage for mapping 0 vs. mapping 1

Fig. 6. Maximal throughput of the system under mapping 0 vs. mapping 1
applications with the node resources, that ensures the best performance. Most of the time, the performance bottleneck in a hardware forwarding system is on the lookup level especially if the application uses several lookups in its processing. We presented through this paper a performance model that will be used for mapping application (e.g. MPLS) tasks with its corresponding network node resources. This work intends to provide a tool that helps the application designer to choose the best mapping while ensuring the best usage of resources and the performance. We used our performance model, with a working version of Openflow 1.1 over network processor, in order to resolve the lookup bottleneck of the OpenFlow 1.1 multiple tables by finding the best memory mapping instead. Experimentation results obtained by either simulation or with real hardware shows that the best mapping is the one that regroups the lookups into the same processing element (PE) and therefore provide less parallel load to the node memory.

The evaluation of the performance model proposed in this paper is done with MPLS applications. However, it can be used to map other type of applications with the hardware resources. Multi-path and tunnels, for instance, are examples of applications supported by OpenFlow 1.1 that can be useful in the context of data center networks.

REFERENCES