A Fast Network Processor Performance Analysis Approach

Hao Che, Basavaraj Menasinahal, Chethan Kumar

Department of Computer Science and Engineering
University of Texas at Arlington
(hche@cse.uta.edu, basavaraj_m@yahoo.com, chethan@uta.edu)

Abstract

How to allow fast network processing unit (NPU) performance testing in supporting fast data path functions in a router is a challenging issue. It is more so in a router design phase when there are a vast number of design choices to be tested and the microcode for the fast data path functions is yet to be developed. In this paper, based on the instruction-and-latency-budget-based NPU analysis methodology, we put forward an approach to allow NPU throughput upper bounds at arbitrary number of threads to be estimated quickly (in a fraction of a second on a Pentium II PC). These performance bounds allow the performance of fast data path functions to NPU configuration mapping to be quickly tested solely based on the worst-case code path derivable from the pseudo code of the fast data path functions. Case studies based on the code samples available in the Intel IXP 1200 and 2400 Developer Workbenches are performed. The performance bounds are found to be within 17% of the cycle-accurate simulation results.

Key Words: Network Processor, Multithreaded Processor, Performance Analysis and Design Aids, Network Protocols

1. Introduction

As the Internet applications proliferate, network processing units (NPUs) or network processors have been constantly pushed to their capacity limits in handling an ever growing list of data path functions in a router. It is challenging to program an NPU to enable rich router functions without compromising wire-speed forwarding performance. Even more so is during the router design phase when a router designer or NPU programmer is faced with a vast number of design choices in terms of data path function partitioning among multiple NPUs and data path function mapping to a desired NPU configuration. A misjudgment can lead to either re-designs at various design stages or poor packet forwarding performance. The traditional NPU performance analysis tools, e.g., cycle-accurate simulation tools, are unviable in helping reach a quick decision, especially in a router design phase when the microcode is yet to be developed. Hence, a new approach which can help make such a decision quickly is much needed.

To address the above issue, a widely adopted methodology in practice is to use both *instruction budget* and *latency budget* for NPU performance testing, as documented in [1] (we call it *Instruction-LAtency-Budget-based methodology (ILAB)*). ILAB aims to allow fast estimation of NPU performance when a given (worst-case) code path is mapped to a micro-engine (ME)[1] pipeline stage. The idea behind this approach is to use instruction budget and latency budget, obtained in the worst-case (e.g., when minimum sized packets arrive back-to-back at wire-speed [1]), to test whether the wire-speed can be sustained for the ME pipeline stage. Meeting these two budgets for all the ME pipeline stages ensures wire-speed forwarding performance. On the other hand, failing to meet any of these budgets at any ME pipeline stage guarantees that the wire-speed forwarding performance cannot be achieved. ILAB captures the essence for the NPU performance analysis. However, it is challenging to estimate the total latency a packet spent in an ME pipeline stage for latency budget testing, simply because how long a packet will stay in an ME pipeline stage is a complicated function of the code path, the number of threads, and the thread scheduling algorithm.

In this paper, a novel approach is proposed to allow quick estimation of the packet latency in an ME pipeline stage for the coarse-grain thread scheduling discipline and for any given code path and any given number of threads. This latency estimation further allows the NPU throughput upper bounds to be estimated, making ILAB a powerful means for fast NPU performance testing.

A. Literature Background

[1] Also known as processing element (PE) or nP core.

In line with the traditional approach for the performance analysis of a computer system, NPU modeling and simulation tools were developed, e.g., [2-8]. These tools aimed at faithfully emulating the NPU microscopic processes, and are useful for fine-tuning the NPU configuration for performance optimization. They are not designed to allow fast NPU performance testing. For example, even for the most lightweight NPU simulator described by Xu and Peterson [6], it is reported that it takes 1 hour to simulate 1 second of hardware execution on a Pentium III 733 PC with 128 Mbytes memory, assuming the microcode is available as input to the simulator. Apparently, it would be impractical to use these simulation tools to reach a quick decision on various design choices, especially in a router design phase when the microcode is yet to be developed. Interesting analytical approaches are also developed for NPU performance analysis, e.g., [10-13]. However, these approaches are either not aimed at fast NPU performance testing (e.g., [10], [12]) or cannot be generally applied to code paths with multi-memory accesses (e.g., [11] [13]).

Intel recently developed an architecture tool [9] which addresses the issue similar to the one addressed in this paper, i.e., to allow NPU performance to be tested at an early design stage when the microcode is yet to be developed. However, the architecture tool described in [9] did not explain the methodologies used in the design of the tool and it did not give any performance data either. Moreover, the tool is particularly designed for Intel IXP2xxx processors

As aforementioned, ILAB is documented in [1]. However, due to the use of total memory access latency, rather than the total latency, for latency budget testing, the performance bound found in [1] is independent of the number of threads in use, which generally leads to overly optimistic estimation of the NPU performance, as we shall explain in details later. In this paper, we address this issue by developing a fast latency estimation algorithm that accounts for the threading effect and offers reasonably tight throughput bound for any given number of threads and coarse-grained thread scheduling discipline.

The rest of the paper is organized as follows. Section 2 gives an example to explain the concept of code path and how to identify potential worst-case code paths. Section 3 formally defines the notations and ILAB. Section 4 presents an algorithm for calculating the total latency and throughput upper bounds. Section 5 tests the accuracy of the ILAB and the algorithm in bounding the NPU performance by comparing the bounds with the cycle-accurate simulation results. Finally, Section 6 concludes the paper and proposes future research work.

2. Worst-Case Code Path

In this section, an example is used to explain the code path concept and how a potential worst-case code path can be identified[2]. A user may test multiple code paths one at a time if he/she cannot decide which one is the worst-case code path.

Fig. 1 gives a typical but simplified *fast data path flow diagram* or graphical representation of the pseudo code for the fast data path functions to be processed in an NPU, including IP forwarding, label swapping for multiprotocol label switching (MPLS), and the IS-IS routing protocol processing. Assume the entire flow diagram is mapped to a single ME. An incoming packet is in the form of an Ethernet frame. The NPU first inspects the EtherType subfield in the Ethernet header to identify the upper layer data format in the frame payload. There are four possible outcomes:

a) It is an IS-IS routing protocol packet. In this case, the frame is sent to the control card without further processing;
b) It is an IP packet. In this case, the IP forwarding is performed which may include firewall/policy filtering, Network Address Translation, DiffServ (i.e., Differentiated Services) traffic conditioning, IP forwarding table lookup or equivalently the longest prefix matching (LPM), TTL (i.e., Time-to-Live) update, checksum update, and so on. Then the layer 2 framing is performed on the packet which may include outgoing interface maximum transmission unit (MTU) check, packet fragmentation, address resolution protocol (ARP) table lookup, and the layer 2 header encapsulation.
c) It is an MPLS encapsulated IP packet. In this case, the MPLS label swapping table lookup is performed. As a result, there are two possible outcomes, i.e., the packet needs to be label forwarded or IP forwarded downstream. In the former case, the label is swapped and the layer 2 framing is performed on the labeled packet. In the latter case, the label is popped off and the IP processing/forwarding in case (b) is performed.

[2] Note that, as in [1], in this paper, we assume that the worst-case code path is already known so that the worst-case performance can be tested by loading an ME pipeline stage with packets carrying this worst-case code path. This section is not meant to offer a formal approach for the identification of the worst-case code path, which is out of the scope of this paper.

d) It is an unknown protocol. In this case, the frame is simply discarded.

Note that for simplicity, Fig. 1 did not provide all possible branches involved. For example, for MPLS label swapping, if 3 labels are supported in the label stack, there can be 12 different cases corresponding to the combinations of popping off 1, 2, or 3 labels and pushing 0, 1, 2, or 3 labels in the label stack. Also the IP Forwarding and Layer 2 Framing blocks further involve multiple branches, which are easy to identify but not shown in Fig. 1. Any unique path from the root to a leaf defines a *code path*. Note that when there is a loop, the loop should be unwrapped with different numbers of looping belong to different branches or code paths.

In the above example, one can easily identify the potential *worst-case code paths* to be tested, i.e., the microcode implementation of the longest flow branch with the largest number of instructions and memory accesses as highlighted along the bold arrows in Fig. 1. This branch involves MPLS label swapping (e.g., popping off 3 labels and pushing 0 label) and then IP forwarding and layer 2 framing, corresponding to the scenario when a packet emerges from an MPLS domain and is to be sent into an IP domain.

From an ME's point of view, the worst-case workload takes place when all the packets require the worst-case code path processing. In other words, any other mixtures of the code paths will generate a workload no worse than this worst-case workload. For instance, the second longest code path, i.e., the one for IP forwarding in case (b), is just a subset of the worst-case code path. A combination of this code path and the worst-case code path will create a workload better than the worst-case workload.

3. ILAB: Definitions and Notations

For the ease of discussion, we consider an NPU organization depicted in Fig. 2. There are M_{PL} MEs working in parallel, handling packets from different interfaces/ports, respectively, with a maximum line rate of *R bps* each. Different MEs share a set of on-chip or off-chip resources, e.g., an external DRAM or an external look-aside coprocessor, collectively denoted as *MEM*. Each ME has M_T threads. Each thread can be configured to handle a packet throughout the lifetime of the packet in the NPU. Each ME has a set of embedded resources shared by all M_T threads, collectively represented by *Mem*. Each thread also has its own set of resources, collectively denoted as *mem*. Note that resources *MEM*, *Mem*, and *mem* only include those which will cause the stall of a thread if accessed by that thread and consequently a context switching. In this paper, the access of any of these resources is collectively called an *I/O event* or equivalently an *I/O access*. An example NPU organization, which bears the most resemblance to the one in Fig. 2, is the one for AMCC's NP7120 NPU [14].

Note that although the above NPU organization happens to coincide with a specific NPU architecture, all the results derived based on this organization applies to any NPU organizations. This is because the analysis concerns with individual MEs only. In other words, whether different MEs in an NPU are configured in pipeline, parallel, or a mixture of pipeline and parallel stages, the wire-speed performance is guaranteed if and only if each and every ME meets its own budgets, as also observed in [1]. The effects of all possible resource contentions among threads at various levels are generally captured in terms of *mem*, *Mem*, and *MEM*.

The following terminologies are used throughout the rest of the paper:

Code Path: a unique sequence of instructions to be executed by an ME for a given packet
Unloaded latency: I/O access latency without contention or queuing delay
Loaded latency: I/O access latency with heavy contention or queuing delay
Instruction Budget: the maximum number of cycles or instructions (assume one instruction per cycle) an ME's arithmetic logic unit (ALU) can spend on each packet without compromising the throughput performance under the worst-case traffic load
Latency Budget: the maximum time duration a packet can stay in a ME without compromising the throughput performance under the worst-case traffic load

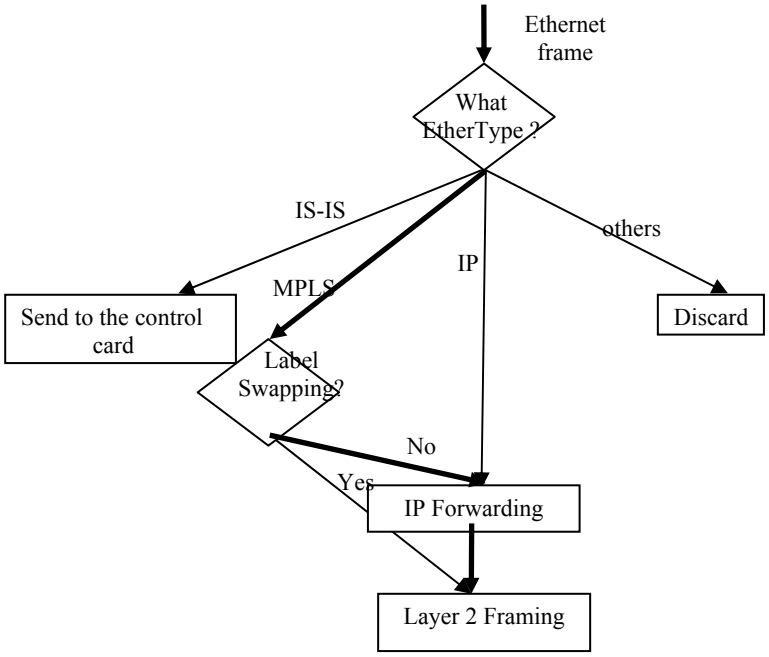

Fig. 1 Fast data path functions

The following parameters are used throughout the paper:

R: line rate in the units of bits per second

K: number of distinct data path flows or code paths

T_P: minimum packet arrival interval in the units of ME clock cycles

P: minimum packet size in the units of bits

F_{ME}: ME clock rate in the units of *Hz*

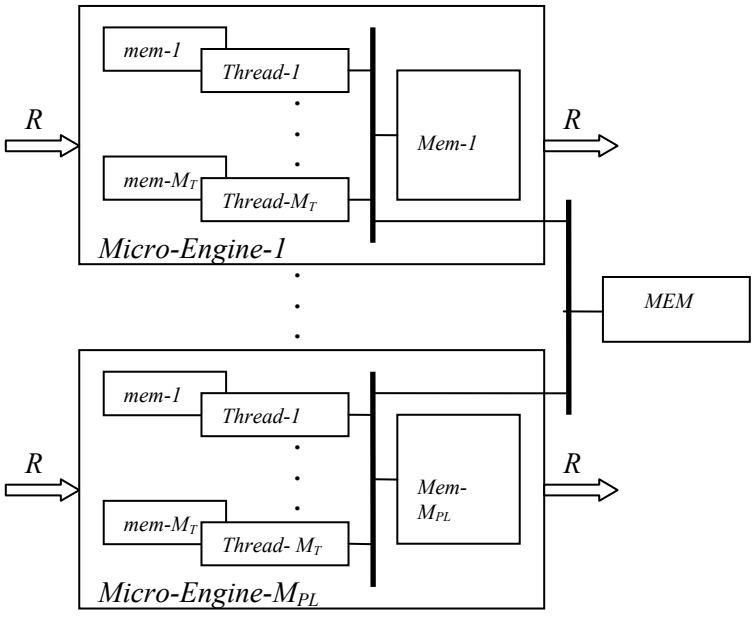

Fig. 2 NPU Organization

M_T: Number of threads per ME

I_{IB}: Instruction budget in the units of ME clock cycles (assuming one instruction per cycle)

L_{LB}: latency budget in the units of ME clock cycles
M_{PL}: Number of MEs working in parallel

The following relationships among different parameters hold for each ME in Fig. 2:

$$T_P = F_{ME}\, P/R, \qquad I_{IB} = T_P, \qquad L_{LB}=M_T\, T_P. \tag{1}$$

To facilitate the analysis, we use the following code path definition:

$T_{l,k}(M_k; m_{1,k}, t_{1,k}, \tau_{1,k}, \ldots, m_{Mk,k}, t_{Mk,k}, \tau_{Mk,k})$: the code path k mapped to ME l with access to resource $m_{i,k} \in mem, Mem,$ *or MEM,* with I/O access latency $\tau_{i,k}$ after the $t_{i,k}$-th cycle in the code path, where $l = 1,2, \ldots, M_{PL}$, $k = 1, \ldots, K$, and $i = 1, 2, \ldots, M_k$, where M_k is the total number of I/O accesses.

$|T_{l,k}|$: the total number of cycles the ALU in ME l spent on the code path $T_{l,k}(M_k; m_{1,k}, t_{1,k}, \tau_{1,k}, \ldots, m_{Mk,k}, t_{Mk,k}, \tau_{Mk,k})$, where $l = 1,2, \ldots, M_{PL}$, and $k = 1, 2, \ldots, K$. See Fig. 3 for a graphical representation of the code path.

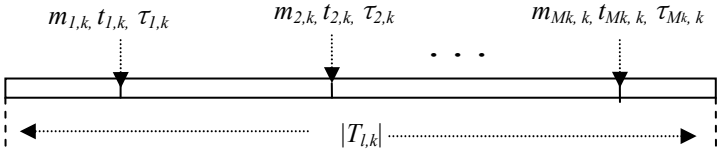

Fig. 3 $T_{l,k}(M_k; m_{1,k}, t_{1,k}, \tau_{1,k}, \ldots, m_{Mk,k}, t_{Mk,k}, \tau_{Mk,k})$

In the code path definition, $\tau_{i,k}$ is the unloaded I/O latency. Note that $|T_{l,k}|$ does not include the latency cycles for I/O accesses. Also, note that the same resource may be accessed more than once. Therefore, different $m_{i,k}$'s may refer to the same resource, accessed at different time instants. Even for the same resource access, different $\tau_{i,k}$*'s* may take different values. This is because different accesses may perform different tasks with different write/read operations. For a memory access with both read and write operations, $\tau_{i,k}$ should represent the whole duration of the process to account for the memory access serialization effect.

The above code path definition only counts the number of instructions and the I/O events and no instruction level details are included. This allows the analysis to be applicable to the NPU programming phase when the micro-code is yet to be developed. However, the lack of instruction level details does come with a cost. Namely, any instruction driven events cannot be accounted for in ILAB. Hence, the ALU time spent on processing ILP aborts needs to be estimated by other means, if necessary. Section 5 will discuss this issue in more details.

We further define $L_{l,k}$, the total latency for the k-th code path in ME l, i.e., the time duration a packet with the k-th code path stays in ME l. We have,

$$L_{l,k} = |T_{l,k}| + \sum_{j=1:\, Mk} (\tau_{j,k} + \tau^{q}_{j,k} + \tau^{W}_{j,k}), \tag{2}$$

where $\tau_{j,k}$ is the unloaded I/O latency, $\tau^{q}_{j,k}$ is the queuing delay due to $m_{j,k}$ access contention, $\tau^{W}_{j,k}$ is the thread waiting time in the ready state after the completion of $m_{j,k}$ I/O access, and $|T_{l,k}|$ is the number of ALU cycles spent on the code path k.

Now, ILAB can be formally expressed as follows:

For a given set of code paths mapped to ME l, wire-speed processing in ME l is achieved if and only if the both instruction and latency budgets for all the code paths are met, i.e.,

$$|T_{l,k}| \le I_{IB} \quad and \quad L_{l,k} \le L_{LB}, \qquad for\ k = 1, 2, \ldots, K, \tag{3}$$

The relationships for the two budgets state that to ensure wire-speed forwarding, the ALU in each ME can spend no more than $I_{IB} = T_P$ cycles on processing each packet and a packet cannot stay in the ME for a time duration longer than $L_{LB}=M_T\, T_P$.

A pseudo code can used to estimate $|T_{l,k}|$ and $L_{l,k}$. Then whether the wire-speed forwarding could be achieved can be easily tested based on Eq. (3). However, while $|T_{l,k}|$ can be estimated fairly accurately by estimating the total number of instructions assuming one instruction per clock cycle, estimating $L_{l,k}$ is much harder, as shall be seen in the next section. Finally note that in [1], L_{LB} is used to bound $\sum_{j=1:Mk} \tau_{j,k}$, the total I/O latency, independent of $|T_{l,k}|$ and $\tau^W_{j,k}$. In fact, L_{LB} should be used to bound $L_{l,k}$ as in (2), which is strongly dependent on the number of threads in use due to $\tau^W_{j,k}$.

4. Performance Bounds

In this section, we develop an algorithm to estimate $L_{l,k}$ and consequently the throughput bounds. We assume that a *coarse-grained thread scheduling discipline* is used in all the MEs, which is the case for Intel IXP series. This discipline allows a thread to be executed continuously until there is an I/O event or a programmer-defined voluntary yielding event. When such an event occurs, the thread stalls and the control is passed to the next thread in a round-robin fashion. For simplicity, it is assumed that the number of ALU cycles spent on a context switching is negligible, which is true for most commercial NPUs (one cycle in general).

As can be seen in Eq. (2), $L_{l,k}$ is composed of four terms. The first term $|T_{l,k}|$ can be estimated when a pseudo code or a flow diagram such as the one in Fig. 1 is available. The second and the third terms in the sum, i.e., $(\tau_{j,k} + \tau^q_{j,k})$ represent $m_{j,k}$ access latency including the unloaded latency and queuing delay. $(\tau_{j,k} + \tau^q_{j,k})$ can be approximated by either the loaded latency as used in [1], or unloaded latency, assuming $\tau^q_{j,k} = 0$ (this is a good approximation when the I/O resource corresponding to $m_{j,k}$ is shared by only a small number of threads and/or the I/O access is pipelined). Hence, from now on, $(\tau_{j,k} + \tau^q_{j,k})$ is replaced by $\tau_{j,k}$, which represents either loaded or unloaded latency. Then the problem is narrowed down to the last term in the sum, i.e., $\tau^W_{j,k}$. In what follows, we propose an approach to estimate the worst-case bound for $\tau^W_{j,k}$.

A simple approach is to use the following worst-case bound:

$$\tau^W_{j,k} \le \Delta\check{T}_k - 1 , \tag{4}$$

$$\text{where } \Delta\check{T}_k = (M_T\text{-}1)\, Max_{\{m=1:Mk+1\}} \{ t_{m,k} - t_{m-1,k} \}, \tag{5}$$

where $\Delta\check{T}_k$ is defined as the worst-case turn-around time, i.e., the worst-case time duration from the instant the next thread becomes active to the instant the thread in question is given a chance to be executed again. Note that in Eq. (5), $t_{0,k} = 0$ and $t_{Mk+1,k}=|T_{l,k}|$. For the coarse-grained thread scheduling discipline, the worst-case turn-around time is when all other $(M_T\text{-}1)$ threads execute the *longest segment* in the code path, i.e., $(Max_{\{m=1:Mk+1\}} \{ t_{m,k} - t_{m-1,k} \})$, and $\Delta\check{T}_k$ is given by Eq. (5). Inequality (4) simply states that the worst-case thread waiting time $\tau^W_{j,k}$ occurs when there is one cycle left for the I/O access when the thread is given a chance to be executed, which leads to a worst-case turn-around time $\Delta\check{T}_k$ before the thread gets another chance to be executed. Let $L^D_{l,k}$ be the estimated latency upper bound, i.e.,

$$L^D_{l,k} = |T_{l,k}| + \sum_{j=1:Mk} (\tau_{j,k} + \Delta\check{T}_k - 1). \tag{6}$$

According to Eqs. (1) and (3), to meet both budgets we must have,

$$|T_{l,k}| \le I_{IB} = T_P, \qquad L^D_{l,k} \le L_{LB} = M_T T_P . \tag{7}$$

This leads to the following estimation of the minimum packet arrival interval,

$$T_P = Max\{|T_{l,k}|, L^D_{l,k}/M_T\} . \tag{8}$$

And according to Eq. (1), the maximum sustainable line rate,

$$R = F_{ME} P / Max\{|T_{l,k}|, L^D_{l,k}/M_T\} \quad . \tag{9}$$

The potential problem with the use of the worst-case bound in Eq. (4) is that the bound can be very loose if the largest code path segment in the worst-case code path is large. To remedy this problem, we first define a thread scheduling discipline called the *Grain Size n* (**GS**(*n*)). **GS**(*n*) is the coarse-grained thread scheduling discipline with the addition of a minimum number of *virtual* voluntary yielding instructions embedded in the worst-case code path to reduce the

maximum segment size to n, where $1 \le n \le Max_{\{m=1:Mk+1\}} \{ t_{m,k} - t_{m-1,k} \}$. Note that unlike the voluntary yielding feature available for Intel IXP series [6], the voluntary yielding trigger here is considered virtual in the sense that it is not a real trigger, but simply a switch of control to the next thread without cost (e.g., cycles for context switching and ILP aborts). The smaller (larger) the n is, the finer (coarser) grained **GS**(n) is. **GS**(n) reduces to the coarse-grained thread scheduling discipline at $n = Max_{\{m=1:Mk+1\}} \{ t_{m,k} - t_{m-1,k} \}$ and it degenerates to the fine grained thread scheduling discipline at $n = 1$. Now, we have the following result:

Theorem 1: *For any given number of threads, if n > n', **GS**(n) can hide no less I/O latency from the ME ALU than **GS**(n').*

Proof: Note that if there is at least one thread available for execution at anytime, it doesn't matter, in terms of the overall throughput performance, whether the ALU executes up to n or n' instructions from one thread at a time, as long as the ALU is busy until all the instructions are executed. In this case, ***GS**(n')* and ***GS**(n)* can be expected to perform equally well in getting the job done. However, executing n' instructions instead of n for $n'<n$ from one thread at a time will increase the chance of putting the ALU in the idle state or in the non-work conserving mode. This is simply because as n gets smaller, the instruction executions for different threads become more parallelized, making their next I/O accesses more synchronized, hence more ALU idle time waiting for I/O events. □

Define $L_{l,k}(n)$ as the total packet latency in ME l when **GS**(n) is used. On the basis of Theorem 3, we have $L_{l,k}(n) \le L_{l,k}(n')$, *if* $n \ge n'$. This leads to the following important result:

Corollary 1: *Any upper bound of $L_{l,k}(n)$ for **GS**(n) is an upper bound of $L_{l,k}$ for the coarse-grained thread scheduling discipline.*

Proof: From $L_{l,k}(n) \le L_{l,k}(n')$, *if* $n \ge n'$, we have $L_{l,k} \le L_{l,k}(n)$, for $n = 1, 2, ..., Max_{\{m=1:Mk+1\}} \{ t_{m,k} - t_{m-1,k} \}$. Hence any upper bound of $L_{l,k}(n)$ must be an upper bound of $L_{l,k}$. □

This result allows us to use the minimum of all the upper bounds (as in Eq. (4)) found for *GS*(n)'s (where $\Delta\check{T}_k = (M_T\text{-}1)n$, for $n = 1, 2, ..., Max_{\{m=1:Mk+1\}} \{ t_{m,k} - t_{m-1,k} \}$) **as the upper bound for $L_{l,k}$.** As we shall see in Section 5, the latency bound found in this way is pretty tight for most cases studied. A less careful thought may lead to the conclusion that the minimum upper bound always occurs at $n=1$, since this would give the minimum turn-around time $\Delta\check{T}_k$. This is simply not true because although $\Delta\check{T}_k$ gets smaller as n reduces, the number of round robins increases due to more frequent context switching as n reduces. Therefore, for any given M_T, there is an optimal n value for which the upper bound for $\tau^{W}_{j,k}$ is the tightest. For any given code path, searching for this upper bound can be easily done numerically. We wrote a C program of less than 30 lines to implement this algorithm. It finds the latency and throughput upper bounds in Eq. (6) and (9), respectively, for any given number of threads in a fraction of a second on a Pentium II PC.

5. Case Studies

In this section, the accuracy of the proposed solution is tested against the cycle-accurate simulation. Due to the page limitation, we only present the testing based on an IP forwarding code sample available in Intel IXP1200 SDK Developer Workbench. In an extended version of this paper [15], we tested a set of code samples available in both IXP 1200 and 2400 simulators, which represent a wide spectrum of data path applications typically seen in a router. The results are consistent with the one presented in this paper. This section is composed of two subsections. In the first subsection, we test the tightness of ILAB or bounds in Eq. (3). In the second subsection, we test the accuracy of the latency and throughput bounds found by the algorithm developed in the previous section.

A. ILAB Testing

To test ILAB, the following simulation using Intel IXP1200 SDK cycle-accurate simulator is performed. One ME with up to four threads is configured for receiving and processing the packets from a single port, and two other MEs with four threads each are configured for transmitting the packets, resulting in a two-stage pipeline configuration. All the packets have the same code path at the receive stage, i.e., an IP forwarding code path, directly taken from a code sample which comes with the simulator (See Table 2 for details). This configuration creates a potential bottleneck at the receive stage in the pipeline, which allows ILAB to be tested, as if only the receive stage exists. Since only a single port is supported, packets arrive sequentially at a fixed time interval of T_P cycles, which determines both I_{IB} and L_{LB} as given in Eq. (3). By increasing the packet arrival rate or reducing T_P (refer to the first equation in Eq. (1)) till there is a packet

drop, one can then identify the maximum or saturated line rate the ME can sustain when a given number of receive threads (from 1 to 4) is configured. The results are shown in Table 1 (with the parameter settings: *ME clock rate = 200 MHz*, *packet size = 64 bytes*, and code path length at the receive stage *=157 instructions or cycles*).

First, note that the ALU time is partitioned among active, aborted, stalled, and idle states. The active state is when the ALU executes the instructions; the aborted state is when ALU deals with broken ILPs due to context switching, branching, etc.; the stalled state is when ALU is in a waiting state; the idle state is when ALU is idle. Second, one notes that the instruction budgets for all four cases are not exceeded (note that the *code path length =157*). Second, the simulated total latencies closely match their respective latency budgets, meaning that the latency budgets are barely met and further increase in the line rate will result in packet losses, agreeing with the simulation results. The results also suggest that for the code path studied, the latency budget be constrained, not the instruction budget. Three more code samples are used in [15] to do the testing and the results are found to be consistent with the above ones.

# Threads (M_T)	**Simulated Saturated Line Rate (R_s) (Mbps)**	**% ALU Active-RX Estimated (η_l=157/I_{IB})**	**% ALU - RX**				**Instruction budget (I_{IB} RX stage)**	**Latency Budget(L_{LB} RX stage)**	**Simulated Total Latency ($L_{l,k}$) (RX stage)**
			Active (α_{active})	Aborted ($\alpha_{aborted}$)	Stalled ($\alpha_{stalled}$)	Idle (α_{idle})			
1	183	28.04	28.8	9.4	0.4	61.4	560	560	537
2	328	50.32	51.4	15.5	1.0	32.1	312	624	600
3	431	65.97	68.7	19.1	1.4	10.8	238	714	687
4	449	68.84	75.6	22.4	1.5	0.5	228	912	872

Table 1 Simulation testing for ILAB

B. Throughput Bound Testing

The IP forwarding code path related information in the receive pipeline stage is listed in Table 2. Since Intel IXP1200 adopts a *store-and-forward architecture*, it requires moving the packet into and out of a SDRAM, writing the packet descriptor to and reading it from a SRAM, and multiple IP lookups in a SRAM, resulting in multiple stalls. The number of instructions for each code path segment $(t_{m,k} - t_{m-1,k})$ and $\tau_{j,k}$ (the unloaded latency in this example[3]) for each I/O access are estimated and listed in columns 2 and 4, respectively. The Task and Type of I/O Access columns are listed for the purpose of completeness and the semantics of instructions and I/O accesses do not play any role in the worst-case analysis, which involves only the code path information in columns 2 and 4.

The testing results are summarized in Table 3. Note that two sets of results are given. The first set (columns 2 to 4) assumes that one has no knowledge about $\alpha_{aborted}$ /α_{active} in Table 1 and the estimation does not take the possible imperfection of ILP into account. The second set (columns 5 to 7) assumes that one has perfect knowledge about $\alpha_{aborted}$ /α_{active}, which is assumed to be the same as the measured one in Table 1, and the imperfection of ILP is accounted for the code path length estimation. For $M_T = 1$, $L^D_{l,k}$ in column 2 is also obtained from Table 2 by simply taking the sum of the two data in the last row. The rest of the results in columns 2 are the tightest bounds found by searching the minimum of the latency bounds of $GS(n)$ for $n = 1, 2, ..., Max_{\{m=2:Mk\}} \{ t_{m,k} - t_{m-1,k} \}$. The corresponding estimated maximum line rate R's in column 3 are calculated from Eq. (9) and their errors with respect to the simulated maximum line rates R_s's as listed in Table 1 are also given in column 4. The same set of data based on the effective $|T_{l,k}|$ with abort effect accounted is given in columns 5 to 7.

One can see that even without taking into account the imperfection of ILP, the estimated line rates closely match the simulated ones within 16% errors. The last column shows that by taking into account the imperfection of ILP, the errors further reduces to about 12% or less. The results for six other case studies based on both IXP 1200 and 2400 SDK simulators in [15] are found to be consistent with the one presented here.

[3] The queuing delay $\tau^q_{j,k}$ due to resource contention for all the I/O accesses are found to be very small for almost all the cases studied and can be neglected.

Task	Instructions between stalls ($t_{m,k} - t_{m-1,k}$)	Type of I/O access	Unloaded latency $\tau_{j,k}$
Check receive ready flags	5	FBI read	14
Move packet from IX Bus to RFIFO	8	FBI write + IX Bus receive	76
Read recive control information (after reading packet from IX Bus to RFIFO)	2	FBI read	19
Wait for buffer allocation (in SDRAM); get the descriptor from SRAM	11	SRAM read	17
Read 2 Quad words from RFIFO into microengine for IP validation	16	RFIFO read	18
Read 2nd 32 byte to SDRAM (in the allocated buffer)	15	RFIFO read	22
LPM (IP lookup)	40	SRAM read	17
LPM (IP lookup)	7	SRAM read	17
LPM (IP lookup)	5	SRAM read	17
Get next hop information from SDRAM	7	SDRAM read	47
Write packet descriptor to SRAM (after associating it with a TX port)	16	SRAM write	18
Read queue descriptor from SRAM (for enqueue operation)	4	SRAM read	22
Write the packet descriptor to SRAM (to the TX queues associated with the TX port)	15	SRAM write	20
Misc	6		
TOTAL	**157**		**324**

Table 2 Worst-case total latency estimation for generic IP forwarding in IXP1200

# of Threads (M_T)	*$L^D_{l,k}$ (aborted not cosidered)*	*R (Mbps) (aborted not considerd)*	*Error(%) ($\|R-R_s\|/R_s$) (aborted not considered)*	*$L^D_{l,k}$ (aborted considerd)*	*R (Mbps) (aborted considered)*	*Error(%) ($\|R-R_s\|/R_s$) (aborted considered)*
1	485	211	0.37	535	191	4.59
2	598	343	7.40	649	316	3.79
3	759	405	15.29	811	379	12.11
4	921	445	10.23	973	421	6.24

Table 3 Testing of the throughput bounds

7. Conclusions and Future Work

In this paper, based on the Instruction-and-LAtency-Budget-based methodology (ILAB), we proposed an algorithm to find the tight latency bounds for any ME pipeline stage. This algorithm allows the performance of the fast data path functions to NPU configuration mapping to be quickly tested solely based on the worst-case code path derivable from the pseudo code of the fast data path functions. Case studies based on the code samples available in the Intel IXP 1200 and 2400 Developer Workbenches are performed. The performance bounds are found to be within 17% of the cycle-accurate simulation results.

A limitation of ILAB is that the user of ILAB is responsible for the identification of the worst-case code path from the pseudo-code and there has been no systematic approach for the worst-case code path identification. Hence, our future research will focus on developing an automatic procedure for the worst-case code path identification.

References

[1] S. Lakshmanamurthy, K. Liu, Y. Pun, L. Huston, and U. Naik, "Network Processor Performance Analysis Methodology," *Intel Technology Journal*, Vol. 6, No. 3, 2002.

[2] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: an Infrastructure for Computer System Modeling, " *IEEE Computer*, Vol. 35, No. 2, 2002.

[3] M. Rosenblum, E. Bugnion, S. Devine, S. A. Herrod, "Using the SimOS Machine Simulator to Study Complex Computer Systems," *Modeling and Computer Simulations*, Vol. 7, No. 1, pp. 78-103, 1997.
[4] E. Kohler, R. Morris, B. Chen, J. Janotti, and M. F. Kaashoek, "The Click Modular Router," *ACM Transactions on Computer Systems*, Vol. 18, No. 3, pp. 263-297, Aug. 2000.
[5] Z. Huang, J. P. M. Voeten, and B. D. Theelen, "Modeling and Simulation of a Packet Switch System using POOSL," *Proceedings of the PROGRESS workshop 2002*, pp. 83-91, October 2002.
[6] Wen Xu and Larry Peterson, "Support for Software Performance Tuning on Network Processors", *IEEE Network, July/August 2003*.
[7] Y. Luo, J. Yang, L. N. Bhuyan, and L. Zhao, "NePSim: A Network Processor Simulator with a Power Evaluation Framework," *IEEE Micro, Special Issue on Network Processors for Future High-End Systems and Applications*, Vol. 24, No. 5, pp. 34 – 44, Sept/Oct. 2004.
[8] P. Paulin, C. Pilkington, and E. Bensoudane, "StepNP: A System-Level Exploration Platform for Network Processors," *IEEE Design & Test of Computers*, Vol. 19, No. 6, pp. 17-26, Dec 2002.
[9] "Advanced Software Development Tools for Intel IXP2xxx Network Processors," Intel White Paper, Oct. 2003.
[10] S. Ramakrishna and H. Jamadagni, "Analytical Bounds on the Threads in IXP1200 Network Processor," *Proceedings of the Euromicro Symposium on Digital System Design*, 2003.
[11] P. Crowley, J. Baer, "Worst-Case Execution Time Estimation for Hardware-assisted Multithreaded Processors", *the 9th International Symposium on High-Performance Computer Architecture*: *the Workshop on Network Processors,* Feb., 2001.
[12] L. Thiele, S. Chakraborty, M. Gries, S. Kiinzli, "Design Space Exploration of Network Processor Architectures," *Network Processor Design: Issues and Practices*, Editors: P. Crowley, M. Franklin, H. Hadimioglu, P. Onufryk, Morgan Kaufmann Publishers, October 2002.
[13] T. Wolf and M. Franklin, "CommBench – a Telecommunications Benchmark for Network Processors," in Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 154-162. April 2000.
[14] P. Lekkas, "Network Processors: Architectures, Protocols, and Platforms," McGraw-Hill Publishers, 2003.
[15] Due to the need for double-blind review, we cannot give the source of this extended version. It will be given after review process finishes. To compensate for the loss of information, the following gives a summarized testing result for the code samples in IXP2400 we studied (three other cases for IPX1200 are not presented here):

Application (M_T =8)	**$L^D_{l,k}$ (aborted not considered)**	***R* (Mbps) (aborted not considered)**	**Error(%) *(|R-Rs|)/Rs* (aborted not considered)**	**$L^D_{l,k}$ (aborted considered)**	***R* (Mbps) (aborted considered)**	**Error(%) *(|R-Rs|)/Rs* (aborted considered)**	***Rs* (Mbps) (aborted considered)**
Diffserv POS	3786	811	12	3710	828	8.4	904
MPLS	3404	519	8.1	3335	530	4.2	553
IPv4 Ethernet	3181	821	5.6	3111	839	0.9	850

Table 4 Testing results for three code samples in IXP 2400 at $M_T = 8$

The code samples studied are DiffServ with POS interface, MPLS, and IPv4 with Ethernet Interfaces. The number of threads is fixed at eight. As one can see, the estimated throughput values are within 12% of the cycle-accurate simulation results, even without taking into account the abort effect.