Fundamental Network Processor Performance Bounds

Hao Che, Chethan Kumar, and Basavaraj Menasinahal

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

Abstract

In this paper, fundamental conditions which bound the network processing unit (NPU) worst-case performance are established. In particular, these conditions formalize and integrate, with mathematical rigor, two existing approaches for finding the NPU performance bounds, i.e., the work conserving condition and instruction/latency budget based approaches. These fundamental conditions are then employed to derive tight memory access latency bounds for a data path flow with one memory access. Finally, one of these memory access latency bounds is successfully used to interpret a peculiar phenomenon found in Intel IXP1200, demonstrating the importance of analytical modeling for NPU performance analysis.

1. Introduction

As the Internet applications proliferate, network processing units (NPUs) or network processors have been constantly pushed to their capacity limits, 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. Hence, new methodologies and techniques which can help make such a decision quickly is much needed.

There have been quite a few simulation tools developed for NPU performance analysis/testing, e.g., [1-7]. These tools are 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 [5], 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.

A practical approach to allow fast NPU performance analysis is to exploit the NPU performance bounds instead of attempting to faithfully capturing the instruction level details. One such an approach is based on the CPU *work conserving condition*. The idea is to identify, for a given code path, how many threads need to be configured in order for the CPU to work under work conserving condition, i.e., fully exploiting the available CPU resource to maximize the processing performance. This technique was widely used for the performance analysis of general multithreaded processor systems, e.g., the papers by Saavedra-Barrera, et. al. [8] and Agarwal [9]. Recently, a few interesting research papers, e.g., the papers by Peyravian and Calvignac [10], and Ramakrishna and Jamadagni [11], successfully applied this technique to identify the optimal number of threads needed to maximize the NPU throughput performance.

Another powerful approach, (call it the instruction/latency budget based approach), proposed by Lakshmanamurthy, et. al. [12]. is to use two measures, i.e., the *instruction budget* and *latency budget*, obtained in the worst-case (e.g., when minimum sized packets arrive back-to-back at wire-speed) as performance bounds, to test whether the wire-speed can be sustained when a given code path is mapped to a micro-engine (ME)[1] 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.

In this paper, we exploit the NPU performance bounds, similar to the approaches taken in [8-12], but in a more rigorous and systematic fashion. There are three major contributions made in this paper. First, fundamental conditions which bound the NPU worst-case performance are established. In particular, the work conserving condition based approach is

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

generalized and integrated with the instruction/latency budget based approach with mathematical rigor. More specifically, by introducing the notion of *wide-sense* work conserving condition, which generalizes the conventional work conserving condition (called in this paper the *strict-sense* work conserving condition), we are able to show that the instruction/latency budget based approach is simply a special case of the work conserving condition based approach. Moreover, the introduction of the wide-sense work conserving condition also allows NPU performance to be characterized when the strict-sense work conserving condition cannot be satisified, thus making our approach a more general one than the existing work conserving condition based approach. Second, on the basis of these conditions, tight memory access latency bounds for a data path flow with one memory access are derived. Third, one of these memory access latency bounds is successfully used to interpret a peculiar phenomenon found in Intel IXP1200, i.e., for a sample code known as Packet Count, available in IXP1200 Developer Workbench, adding more than 2 threads reduces the throughput performance.

The rest of the paper is organized as follows. Section 2 presents the fundamental conditions which bounds the NPU throughput performance. Section 3 derives tight memory access latency bounds for any given wire-speed. Section 4 applies one of the memory access latency bounds to interpret a peculiar phenomenon found in IXP1200. Finally, Section 5 concludes the paper and proposes future work.

2. Fundamental Conditions

In this section, fundamental conditions that bound the NPU throughput performance are derived.

A. NPU Organization

Although all the conditions introduced in this section are concerned with a single ME only, for the ease of discussion in the next section, we describe them in the context of an NPU organization depicted in Fig. 1, which involves multiple *MEs*. 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, such as a ternary content addressable memory (TCAM) coprocessor [13], 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*.

B. Terminologies and Notations

The following terminologies are used throughout the rest of the paper:
Data Path Flow: a unique set of data path functions performed by the NPU on a packet.
Code Path: a complete sequence of instructions to be executed by the NPU on a data path flow
Unloaded latency: Memory access latency with no memory resource contention
Loaded latency: Memory access latency with heavy contention or queuing delay
Instruction Budget: the maximum number of cycles or instructions (assuming one instruction per cycle) an ME's arithmetic logic unit (ALU) can spend on each packet without compromising the throughput performance under a targeted worst-case traffic load
Latency Budget: the maximum time duration a packet can stay in a NPU without compromising the throughput performance under a targeted worst-case traffic load
Strict-Sense Work Conserving: an ME's ALU is busy as long as the workload is nonzero, or equivalently, there is one or more instructions to be executed.
Wide-Sense Work Conserving: the percentage of an ME's ALU active time in any latency budget worth of time interval equals the percentage of ALU time needed to process all the packets arrived during this time interval.

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
$T_{P:}$	targeted minimum (i.e. worst-case) packet arrival interval in the units of ME clock cycles
P:	effective packet size in the units of bits used to measure the line rate at the targeted minimum packet arrival interval T_P
F_{ME}:	ME clock rate in the units of *Hz*

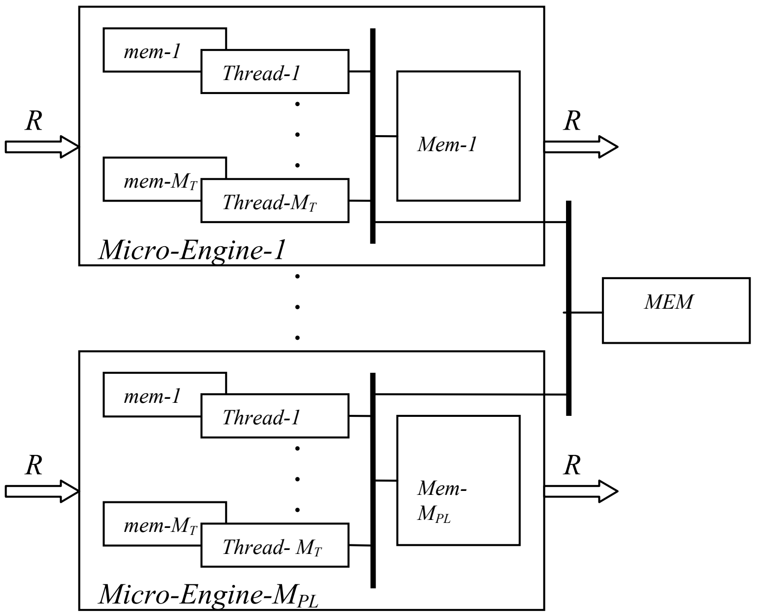

Fig. 1 NPU Organization

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 parameters are also defined:

$C_{l,k}$: The *k*-th code path mapped to ME *l*

$|C_{l,k}|$: The length of the *k*-th code path mapped to ME *l*, i.e., the total number of ME cycles the ALU has to spend on the code path $C_{l,k.}$

$L_{l,k}$: 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*.

$s_{k,l}$: The number of threads which are currently handling the code path *k* in ME *l*.

The following relationships among different parameters hold for the NPU organization in Fig. 1:

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

T_P is a user-defined parameter. If the targeted worst-case packet arrival process is defined as the minimum sized packet arriving back-to-back, T_P represents the minimum packet time and *P* stands for the minimum packet size. Roughly speaking, 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 more rigorous description on when and how to use these two budgets to bound the NPU performance will be explained shortly.

Finally, the required percentage of ALU active time η_l in the worst-case for ME *l* to work under the wide-sense work conserving condition is defined as follows:

$$\eta_l = max_{\{Sk,l\}}\{\ \eta_l(\{s_{k,l}\})\} \tag{2}$$

$$\text{and}\ \ \eta_l(\{s_{k,l}\}) = \textstyle\sum_{k=1:K} s_{k,\,l}\,|C_{l,k}|\,/L_{LB},\ \ \sum_k s_{k,\,l} \le M_T,\ \ for\ l = 1, 2, \ldots, M_{PL},$$

$\sum_{k=1:K} s_{k,\,l}\,|C_{l,k}|$ is the total workload (i.e., the total instruction load) on the ALU that has to be processed in the time duration L_{LB} to keep up with the line rate. $\eta_l(\{s_{k,l}\})$ is a function of the mixture of the code paths $\{s_{k,l}\}$ and η_l is the largest $\eta_l(\{s_{k,l}\})$, or the required percentage of ALU active time in the worst-case, in order to keep up with the line rate.

Hence, if $\eta_l > 1$, the ALU will be overloaded in the worst-case and the wire-speed cannot be sustained. The mixture of the code paths is said to be *nonrestrictive* if the values $s_{k,l}$ may take are nonrestrictive except $\sum_k s_{k,\,l} \leq M_T$, otherwise, it is said to be *restrictive*. A possible scenario where a restrictive mixture of the code paths may occur is when an NPU supports two ports with each having a different set of code paths. In this case, a code path k that exists for one port but not for the other may never take over all the threads, i.e., $s_{k,l} < M_T$.

C. Wire-Speed Forwarding Conditions

Satisfying the strict-sense work conserving condition allows memory access latencies to be completely hidden from the ALU, thus fully exploiting the ALU power to maximize the throughput performance. The following theorem states that under what conditions ME l can achieve the highest processing performance:

Theorem 1: *The maximum sustainable throughput performance for a given set of code paths mapped to ME l is achieved if ME l works under the strict-sense work conserving condition and* $\eta_l = 1$.

Proof: Note that $\eta_l = 1$ means that the ME ALU has just enough processing power to process the offered workload. The strict-sense work conserving condition means that the ME ALU processing power can be entirely devoted to process the offered workload. Hence, the wire-speed can be achieved. Moreover, since the ALU processing power has been exhausted at $\eta_l = 1$, this wire-speed is the maximum throughput the ME l can sustain. □

Note that the existing work conserving condition based approaches, e.g., [8-11], are based on the above strict-sense work conserving condition. However, the strict-sense work conserving condition above is not sufficient for NPU performance analysis for two reasons. First, the strict-sense work conserving condition may not always be attainable. This is because (1) the number of configurable threads M_T is always finite for any NPUs and there is no guarantee that the memory access latencies can be completely hidden even with all M_T threads in use; (2) for certain types of code paths, e.g., a code path with serialization effect, memory access latencies cannot be completely hidden, no matter how many threads are configured (see Section 4 for such an example). Second, in many cases, one is not interested in knowing the maximum sustainable throughput performance, but rather whether or not the wire-speed can be sustained provided that a given number of threads are in use.

Now, the following theorem states that under what conditions a given wire-speed can be sustained, which exactly addresses the above drawback the traditional work conserving condition based approach suffers from:

Theorem 2: *For a given set of code paths mapped to ME l, the wire-speed processing is achieved if ME l works under the wide-sense work conserving condition and* $\eta_l \leq 1$.

Proof: Note that $\eta_l \leq 1$ means that the ME ALU has enough processing power to process the offered workload. The wide-sense work conserving condition means that the ME ALU can finish processing all the offered workload that may arrive in the L_{LB} worth of time interval. This ensures that the ME always has a free thread to receive the incoming packet in the worst-case when packets arrive at the T_P time interval. Hence, the wire-speed forwarding performance can be achieved. □

Both Theorem 1 and 2 apply to both restrictive and nonrestrictive mixture of code paths. Now we show that the instruction/latency budget based approach proposed in [12] is, in fact, a special case of Theorem 2 under the condition that the mixture of code paths is nonrestrictive. The following corollary states the approach:

Corollary 1: *For a given set of nonrestrictive code paths mapped to ME l, wire-speed processing in ME l is achieved if both instruction and latency budgets for all the code paths are met, i.e.,*

$$|C_{l,k}| \leq I_{IB} \quad and \quad L_{l,k} \leq L_{LB}\,, \qquad\qquad for\ k = 1, 2, ..., K, \tag{3}$$

Proof: This corollary is a special case of Theorem 2. First, we show that the condition $\eta_l \leq 1$ in Theorem 2 degenerates to the first inequality in Eq. (3) if the mixture of the code paths is nonrestrictive. Without loss of generality, assume $|C_{l,k}| \geq |C_{l,k+1}|$ for all $k = 1,..., K - 1$. Due to the nonrestrictive assumption, let $s_{1,\,l} = M_T$ and $s_{k,\,l} = 0$, for all $k = 2,...,K$, and $l = 1, 2, ..., M_{PL}$. Then from (1), we have $|C_{l,1}| \leq I_{IB}$, implying that $|C_{l,k}| \leq I_{IB}$, for $k = 1, 2, ..., K$.

Second, we note that the second inequality in Eq. (3) ensures that the ME ALU works under the wide-sense work conserving condition. This is true simply because if the second inequality in Eq. (3) holds, it implies that the ME ALU would be able to finish processing all the workload arrived in the L_{LB} worth of time interval, i.e., the ME works under the wide-sense work conserving condition. Hence, according to Theorem 2, the wire-speed forwarding performance is achieved. □

Note the conditions in Corollary 1 are necessary and sufficient conditions for wire-speed forwarding if the mixture of code paths is nonrestrictive. *For the restrictive case, they are sufficient conditions but not necessary ones, meaning that Corollary 1 may lead to overly conservative performance estimation when the mixture of code paths is restrictive.* Although it is less general than Theorem 2, Corollary 1 is particularly useful for finding analytical bounds. For example, by mapping the worst-case data path flow to a specific NPU configuration, a pseudo code can be constructed and used to estimate $|C_{l,k}|$ and $L_{l,k}$. Then whether the wire-speed forwarding could be achieved can be tested based on Corollary 1. In an extended version of this paper [14], an analytical bound for $L_{l,k}$ is derived. This bound is found to be within 17% of the cycle-accurate simulation results for a large number of code samples available in IXP1200 and 2400 Developer Workbenches. However, due to the page limitation, those results are not presented in this paper.

3. Tight Memory Latency Bounds

The latencies for memory access have been Achilles Heel for NPU to meet the tight time window T_P for packet processing. For example, at the OC48 full line rate, the time window to process a packet can be as small as *40 ns* when 49 bytes minimum sized packets arrive back-to-back. This puts a tight constraint on the memory access latency to achieve wire-speed forwarding performance. For this reason, finding tight memory access latency bounds is of paramount importance for NPU performance testing. A back-of-envelope calculation of the memory access latency bound sometimes can be proven very helpful in making a quick decision on a particular design choice. In this subsection, we use a simple example to demonstrate how the fundamental conditions proposed in Section 2 can be used to exploit the tight memory access latency bounds.

Consider the non-restrictive mixture of code paths. In this case, we have a unique worst-case code path $C_{l,k}$ in the sense that $\eta_l(\{s_{k,l}\})$ in Eq. (2) will be maximized when $s_{k,l} = M_T$. Further assume that this worst-case code path involves only one memory access with loaded latency $\tau_{l,k}$, which results in a context switching. A code path with a single memory access may happen for an NPU based on a *cut-through switching* architecture, such as AMCC nP7120. Different from a *store-and-forward* NPU such as IXP1200, a cut-through switching NPU processes a packet on the fly without moving the packet from the receive buffer to the main memory. An example scenario is when the worst-case code path is for IP forwarding which involves only one memory access to do IP forwarding table lookup, e.g., through a look-aside TCAM coprocessor. It may also occur for a store-and-forward NPU such as IXP1200 at ME pipeline stages other than the receive stage. Based on Theorems 1, 2, and Corollary 1, the following corollary gives the maximum tolerable $\tau_{l,k}$ while maintaining the maximum sustainable NPU throughput performance:

Corollary 2: *Assuming the mixture of code paths is nonrestrictive, and the worst-case code path $C_{l,k}$ involves one context switching due to a single memory access with loaded latency $\tau_{l,k}$, the maximum sustainable NPU throughput performance is achieved if and only if $T_P = |C_{l,k}|$ and*

$$\tau_{l,k} \le (M_T-1)\, T_P, \qquad \text{for } m_{l,k} \in mem,\ Mem,\ or\ MEM. \tag{4}$$

Proof: First, note that according to Theorem 1, to achieve the maximum sustainable throughput performance, an ME must work under the strict-sense work conserving condition, i.e., $\eta_l = 1$ or $T_P = |C_{l,k}|$. Further, note that at $\eta_l = 1$, the wide-sense work conserving condition is equivalent to the strict-sense work conserving condition because the ALU in ME l (for $l = 1, 2, ..., M_{PL}$) must always be active processing the workload at $\eta_l = 1$. This means that Theorem 2 degenerates to Theorem 1 at $\eta_l = 1$. Second, due to the nonrestrictive mixture of code paths assumption, Corollary 1 is equivalent to Theorem 2 and hence it also degenerates to Theorem 1 at $\eta_l = 1$. This leads to the conclusion that the maximum sustainable NPU throughput performance is achieved if and only if the second inequality in Eq. (3) holds (note that the first inequality is automatically satisfied due to the condition: $T_P = |C_{l,k}| \equiv I_{IB}$). Furthermore, since $L_{l,k} = |C_{l,k}| + \tau_{l,k} + \tau^W_{l,k}$, where $\tau^W_{l,k}$ is the thread waiting time for execution after the memory access, the second inequality in Eq. (3) can be written as,

$$\tau_{l,k} + \tau^W_{l,k} \le L_{LB} - |C_{l,k}| = M_T T_P - T_P = (M_T-1)\, T_P.$$

Since the waiting time $\tau^W_{l,k} \ge 0$, one can always maximize $\tau_{l,k}$ by letting $\tau^W_{l,k} = 0$, which leads to (4). □

In general, the loaded memory latency $\tau_{l,k}$ is a function of the unloaded latency $\tau^{u}_{l,k}$, and the maximum possible number of threads m contending for the memory resource $m_{l,k}$, i.e., $\tau_{l,k}=\tau_{l,k}(\tau^{u}_{l,k},m)$. The exact format of this function depends on the detailed memory resource access technology in use. In what follows, we consider a simple memory access model and a thread scheduling discipline for which the tight bounds on $\tau^{u}_{l,k}$ can be explicitly derived.

We assume that the queuing delay for the memory access is *additive*, meaning that if two threads simultaneously attempt to access the memory resource, one thread will have to wait $\tau^{u}_{l,k}$ time units in a queue before accessing this memory resource. We also assume that a coarse-grained thread scheduling discipline is in use, which is the case for Intel IXP series. This discipline allows a thread to be executed continuously until there is a memory 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. Then we have the following results,

Corollary 3: *With a coarse-grained thread scheduling discipline, and assuming that the mixture of code paths is nonrestrictive and there is a worst-case code path $C_{l,k}$ involving one context switching due to a memory access $m_{l,k}$ with unloaded latency $\tau^{u}_{l,k}$, and assuming that the queuing delay for memory access is additive, the maximum sustainable throughput performance is achieved if and only if $T_P = |C_{l,k}|$ (i.e., $\eta_l = 1$) and*

$$\tau^{u}_{l,k} \leq (M_T - 1)T_P \qquad \text{if } m_{l,k} \in mem \tag{5}$$
$$\tau^{u}_{l,k} \leq T_P \text{ and } M_T \geq 2 \qquad \text{if } m_{l,k} \in Mem \tag{6}$$
$$\tau^{u}_{l,k} \leq T_P/M_{PL} \text{ and } M_T \geq 2 \qquad \text{if } m_{l,k} \in MEM \tag{7}$$

Proof: Since, to achieve the maximum sustainable NPU throughput performance, the ALU in ME l has to be active all the time, i.e., $\eta_l = 1$ (for $l = 1, 2, ..., M_{PL}$), the following condition must be met: **No more than $(M_T - 1)$ threads from any given ME can be stalled simultaneously.** For the time being, we assume that this condition is met. Then, based on the additive queuing assumption, we have $\tau_{l,k} = m\tau^{u}_{l,k}$, where $m = 1$, $M_T - 1$, and $(M_T - 1)M_{PL}$, for $m_{l,k} \in$ *mem, Mem, and MEM,* respectively. Note that for $m_{l,k} \in mem$, $\tau_{l,k} = \tau^{u}_{l,k}$, regardless whether the condition is met or not. Substituting these $\tau_{l,k}$ functions into Eq. (4), we arrived at Eqs. (5) – (7).

Now the question is whether the above condition can be met or not. The answer is that for the coarse grained scheduling discipline, the above condition is met when the inequalities in Eqs. (5) – (7) hold. The key is to realize that by using the coarse-grained thread scheduling algorithm, the time interval between two successive stalls for two different threads is equal to T_P, independent of when the memory access occurs during the execution of $C_{l,k}$. This is simply because, for any two successive packet arrivals having the identical code path of size T_P and for the coarse-grained thread scheduling discipline, the time interval between the two memory events from the two identical code paths is T_P, independent of when the memory access takes place in the code path. Hence, for $m_{l,k} \in mem$, this implies that even for $\tau^{u}_{l,k} = (M_T - 1)T_P$, there can be at most $M_T - 1$ threads waiting for memory events, and the ALU always has one thread to be executed, i.e., the memory latency is completely hidden from the ALU. Hence, the condition is met when Eq. (5) holds.

To show that the condition is met when Eq. (6) holds, let $M_T = 2$ and follow the arguments for Eq. (5), it is not difficult to realize that even for $\tau^{u}_{l,k} = T_P$, there can be only one thread stalled at any time, and the other thread is being processed by the ALU.

Again, to show that the condition is met when Eq. (7) holds, letting $M_T = 2$ and following the idea in the proof of Eq. (6), we realize that from each ME, there is exactly one memory access in any T_P time interval in the worst-case and the other thread is being processed by the ALU. □

From Corollary 3, two interesting observations can be made. First, for shared memory resources like *Mem* and *MEM*, the access latency must be "hard" upper bounded to achieve the maximum throughput performance and adding more than 2 threads do not help in hiding longer memory latency. In contrast, for a dedicated memory resource as *mem*, longer access latency can be hidden if more threads are added. Although the assumptions made in Corollary 3 may not be entirely realistic in practice, it does provide significant insights on under what conditions adding threads can help hide more memory latencies and under what conditions it cannot. In the following section, Corollary 3 is successfully used to explain a phenomenon found in the IXP1200, which demonstrates the importance of analytical modeling for NPU performance analysis.

4. Why Adding More Than 2 Threads Harmful?

Unlike all the other code samples from IXP1200 Developer Workbench we studied in [14] where adding more threads improves the throughput performance at the receive stage, Packet Count code sample results in the highest throughput performance at 2 threads beyond which the throughput performance drops. This peculiar phenomenon is also observed in [15] but not well explained. On page 121 in [15], it says: "A little more analysis shows that the issue is with the application, not the multithreading". Then it goes on by saying: "So are two threads always better than four? Sure if, all you want to do is count packets on the IXP12xx. Otherwise, you need to use your understanding of thread contention within a micro-engine to determine what is best for your application." As a matter of fact, this phenomenon is a natural consequence of the existence of a *critical section* at the receive stage and can be well explained by means of Eq. (6) in Corollary 3. In what follows, we first explain what the critical section is. Then we discuss how we may use Eq. (6) to explain this phenomenon.

A critical section[2] is a region in a code path which can be executed by only one thread at a time. A critical section can be a BUS, a buffer, an external port or a control register access (as is the case at the receive stage). Even if there is a context switching from one thread executing the critical section to another thread waiting to execute the critical section, the second thread cannot execute the critical section until the first thread has completed its execution of the critical section. More specifically, for a thread currently executing a critical section, when switches context for a memory event, it still has the control over the critical section and any other thread waiting to execute critical section has to wait till that thread gives up its control over the critical section. This not only introduces serialization effect but also adds extra instructions because the waiting threads for the critical section access have to poll a control register in an attempt to gain control of the critical section. However, if there is a thread waiting to execute any code segment other than the critical section, it will be able to execute its code segment when it gets its share of processor cycles.

Now we consider an extreme case. Namely, the entire code path does nothing but just receives a frame from the input port. In this case, almost the entire code path belongs to the critical section, which does several memory accesses to load the entire frame into the NPU with unsuccessful context switching between memory accesses. Since the thread executions have to be serialized in the critical section, one may view the entire critical section roughly as a single memory access with additive queuing delay. This is very much like the code path studied in Section 3 with $m_{1,k} \in Mem$ and with additive queuing delay. Hence, Eq. (6) in Corollary 3 holds, meaning that adding more than 2 threads cannot improve the throughput performance. Moreover, the throughput must reduce as the number of threads increases from 2 to 3 and then to 4. This is simply because the added instructions used for polling the control register increases as the number of threads increases and they do not do any useful work but simply waste the ALU resource.

One can imagine that as the percentage of the code path belonging to the critical section reduces, the serialization effect becomes less a problem and the performance impact due to the critical section reduces. That explains why we only see this phenomenon for the Packet Count code path in which the critical section constitutes 40% of the total code path.

5. Conclusions and Future Work

In this paper, fundamental conditions were derived, which generalize and integrate the traditional work conserving condition and instruction/latency budget based approaches. These conditions set the foundation upon which fast NPU performance analysis algorithms and tools can be developed. In particular, we demonstrated, based on a simple code path, how these conditions may be used to derive useful memory access latency bounds for wire-speed forwarding. We also demonstrated how useful the analytical bounds are in terms of understanding the NPU behaviors.

With the fundamental conditions developed in this paper, various NPU performance bounds can be exploited and used for both fast NPU performance analysis as well as NPU design space exploitation. Currently, a fast NPU performance simulation tool is being developed based these conditions. The aim is to allow NPU performance bounds to be identified by fast simulation without detailed instruction level information. Another direction of research is to exploit memory latency bounds under various thread scheduling disciplines.

[2] Critical section is a term used by Intel for their IXP series of NP microcode.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] Wen Xu and Larry Peterson, "Support for Software Performance Tuning on Network Processors", *IEEE Network, July/August 2003.*

[6] 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.

[7] 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.

[8] R. H. Saavedra-Barrera, D. E. Culler, and T. Eicken, "Analysis of Multithreaded Architectures for Parallel Computing," *Proceedings of the second annual ACM symposium on Parallel algorithms and architectures*, pp. 169-178, 1990.

[9] A. Agarwal, "Performance Tradeoffs in Multithreaded Processors," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 5, pp. 525 -539, Sept. 1992.

[10] M. Peyravian and J. Calvignac, "Fundamental Architectural Considerations for Network Processors," *Computer Networks Journal*, Vol. 41, pp. 587-600, 2003.

[11] S. Ramakrishna and H. Jamadagni, "Analytical Bounds on the Threads in IXP1200 Network Processor," *Proceedings of the Euromicro Symposium on Digital System Design*, 2003.

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

[13] Z. Wang, H. Che, M. Kumar, and S. Das, "CoPTUA: Consistent Policy Table Update Algorithm for TCAM Without Table Lock," *IEEE Transactions on Computers*, vol. 53, no. 12, pp. 1602-1628, Dec. 2004.

[14] H. Che, C. Kumar, B. Menasinahal, "A Fast NPU Performance Analysis Methodology," available online at: http://crystal.uta.edu/~hche/PUBLICATIONS/publication.htm

[15] E. Johnson and A. Kunze, "IXP 1200 Programming", Intel Press.