Pythia: Scheduling of Concurrent Network Packet Processing Applications on Heterogeneous Devices

Giannis Giakoumakis, Eva Papadogiannaki, Giorgos Vasiliadis and Sotiris Ioannidis
ggiakoum@ics.forth.gr, epapado@ics.forth.gr, gvasil@ics.forth.gr, sotiris@ics.forth.gr
Institute of Computer Science, Foundation for Research and Technology - Hellas

***Abstract*—Modern commodity computing systems are composed of a number of heterogeneous processing units, each one with its own unique performance and energy characteristics. However, the majority of current network packet processing frameworks targets only one device (either the CPU or an accelerator), leaving the remaining computational resources underutilized or even idle. In this paper, we propose an adaptive scheduling approach for network packet processing applications that exploits any heterogeneous architecture that can be found in a commodity high-end hardware setup. Our scheduler not only distributes the workloads to the appropriate devices in the system to achieve the desired performance results, but also enables the multiplexing of diverse, concurrently executed network packet processing applications, eliminating the interference effects introduced at run-time. The evaluation results show that our scheduler is able to tackle any interference in the shared hardware resources as well to respond quickly to dynamic fluctuations (e.g., application overloads, traffic bursts, infrastructural changes, etc.) that may occur at real time.**

I. Introduction

The advent of high-end commodity heterogeneous systems (i.e. systems that utilize multiple processing units, typically CPUs and/or GPUs) has motivated the networking community to exploit alternative architectures [1]–[4]. Yet, the majority of those works often target a single device, usually underutilizing the rest of them. Developing a network packet processing application framework that can exploit any available device efficiently and consistently, between a wide range of diverse workloads running concurrently, is highly challenging. First, interference between different devices needs to be minimized in an automated way [5]. Second, support for multiple, concurrently-running, applications should be provided, which is typical in networking middleboxes. Third, data heterogeneity needs to be considered, as the traffic variability can significantly affects the system's utilization and performance [6]–[8].

In this paper, we propose a scheduling approach tailored for network packet processing workloads executed concurrently in a heterogeneous system. Specifically, our proposed solution is designed to explicitly tackle the heterogeneity that is introduced in the underlying hardware architectures, the applications and the network traffic rate. The scheduler dynamically adapts to performance fluctuations that may occur, such as traffic bursts or overloads. The *contributions* of this work are the following: (i) Performance characterization and power consumption of several typical network applications concurrently executed on heterogeneous, commodity multi-device systems. (ii) A software-based energy profiling tool that reports live power consumption measurements for any device in a commodity system setup, by exploiting the corresponding hardware registers. (iii) A scheduling approach that can efficiently select the best device(s) to execute one or more typical packet processing applications, based on current system and network conditions, using a predefined policy goal.

978-1-7281-5684-2/20/$31.00 ©2020 IEEE

II. System Setup

a) Hardware Setup: Our hardware setup consists of an Intel Core i7-8700K CPU packed with an integrated UHD Graphics 630 GPU and a high-end NVIDIA GeForce GTX 1080 Ti GPU [1]. Our setup presents interesting trade-offs: even though the integrated GPU has fewer and less powerful resources when compared to a high-end discrete GPU or the CPU, it consumes much lower power. It is also directly connected to the system's main memory via a fast on-chip ring bus, which results to fewer data transfers and hence lower processing latency than a discrete GPU. Yet, the CPU is considered the best option for latency-aware setups, as it can sustain very small processing times compared to the batch-oriented processing followed by both GPU types. Our machine is also equipped with a 40-Gbps NIC (4x 10 Gbps ports).

b) Applications: We have implemented the following packet processing applications, using the OpenCL 2.1 SDK:

Deep Packet Inspection (DPI): A very common operation when processing network traffic. We use the Aho-Corasick algorithm, which offers multi-pattern searching, and fed it with 10,000 fixed-string patterns from Snort IDS [9].

Packet Hashing (MD5): Typically used in redundancy elimination and in-network caching systems [10]. We implement the MD5 algorithm, which offers low collisions and is mainly used for checking data integrity or deduplication.

Encryption (AES): We use the Cipher Block Chaining (CBC) operation alongside with a 128-bit key per connection. Due to its nature, this encryption technique is a representative form of computational-intensive packet processing.

III. Implementation

Each of the three applications is implemented as a unique kernel. In OpenCL, an instance of a kernel is called *work-item* and a set of multiple work-items is called *work-group*.

[1]In this work we use an i7 CPU to take advantage of the integrated GPU packed in the same processor die, instead of having a NUMA setup with Xeon processors.

Typically, GPUs contain a very fast thread scheduler, thus it is recommended to spawn a large number of work-groups. In contrast, CPUs perform more efficiently, when the number of work-groups is close to the number of the available cores. Discrete, GPUs have a dedicated memory space, meaning that an explicit data transfer from the host (i.e. CPU) to the device (i.e. GPU) must precede. On top of that, a data buffer, which is required for the execution of a computing kernel, has to be created and associated with a specific OpenCL *context*. Even though different contexts cannot share data directly, the data transfers (host-device-host) and the GPU execution are performed asynchronously, which significantly improves parallelism. After careful evaluation, we notice that data transfer requirements differ per application. For instance, DPI and MD5 kernels do not change packet headers or payloads, so there is no need to transfer them back to the host after the execution. On the other hand, AES kernel changes the packet contents, making backward transfers inevitable. Still, when the processing is performed on the main processor or an integrated GPU, expensive data transfers are not required (both devices have direct access to the host memory) as long as the corresponding memory buffers are explicitly mapped, via the `clEnqueueMapBuffer()` function.

a) Batch Processing: A typical approach is to place packets into batches exactly in the same order they are received through each NIC. However, when using multiple processing devices, packets can be reordered. To prevent reordering, devices are being syncronized using a barrier, which enforces them to execute in a lockstep fashion. There is a major performance drawback when using this approach though, as fast devices have to wait for the slow ones. To bypass this problem, we pre-classify incoming packets by building the typical 5-tuple flows before creating the batches and then enqueue all packets of a flow in the same batch.

b) Performance Measurements: We now present the performance achieved in our hardware setup. We use netmap [11] to generate and transmit network packets to our machine[2]. Due to space constraints, we select to present a fraction of configurations that clearly show the diversity of performance characteristics of each device and application. Each configuration was active for a 10-second window, during which the performance of the system was being monitored every second. Tables I, II and III present both the individual and the aggregated performance achieved by the DPI, AES, and MD5 applications, when executed either standalone or by sharing the device with 1 or 3 co-workers. The same benchmark executions are repeated for all the available devices of the system, i.e. CPU, integrated GPU and discrete GPU. We note that the current implementation of our scheduler supports the concurrent execution of every network packet processing application combined; for the purposes of simplicity though, we present only the combination of two different applications in each device at a time.

[2]Even though our machine is equipped with a 40 Gbps NIC, the overall throughput achieved is not higher that 30 Gbps; the reason is that both our NIC and discrete GPU (GTX 1080 Ti) run at a reduced I/O bandwidth (PCIe x8), due to motherboard PCIe constraints

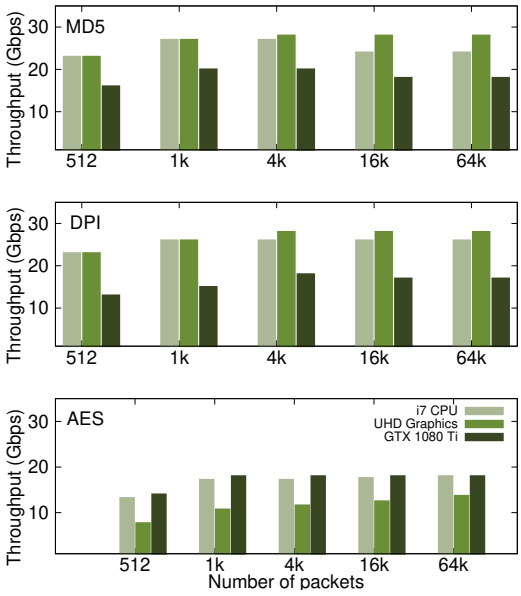

Fig. 1. Throughput sustained for processing 1500-bytes network packets.

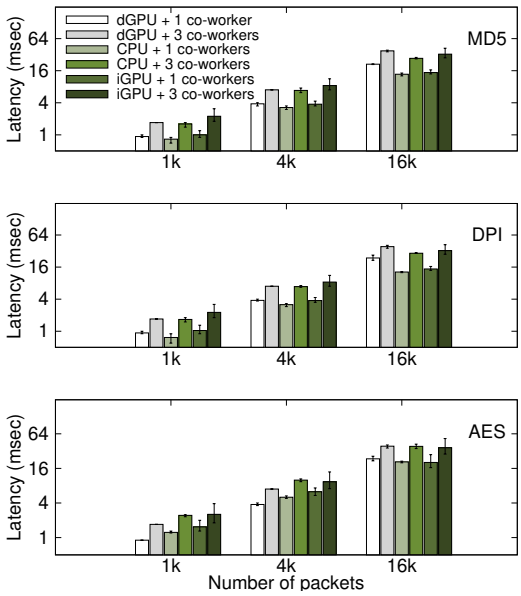

Fig. 2. Overall latency for processing 1500-bytes network packets.

We observe that the benefits from constantly increasing the batch size stop at some point. However, different applications on different devices may require batch size optimizations within a specific range to reach maximum throughput. In the case of DPI, for example, increasing the batch size further of 4096 packets has little impact on the throughput of a discrete GPU. Furthermore, we also notice that the sustained throughput is not consistent across diverse devices. For instance, an integrated GPU seems to be a reasonable choice when performing MD5 and DPI on large packet batches, compared to AES, where the same device results to low

throughput. Overall, the CPU is the best option for latency-aware environments, especially when combined with small batch sizes and not many interfering kernels (Figure 2). Due to its characteristics, AES is an exception to the rule, benefiting more if placed on the discrete GPU even in the case of latency-critical scenarios, regardless of the number of co-workers.

Apparently, there is no clear ranking between the devices, not even a clear winner. A device can actually be the best fit for some applications and the worst fit for some others.

When executing concurrently more than one network packet processing applications in one device, we face the challenge of unknown interference effects, due to contention for hardware resources, software resources and false sharing of cache blocks. In the case of the GTX 1080 Ti, for example, we can see that a large batch size (16K) has negative effects in cases where more than one applications are being executed. The reason is that both the discrete graphics card and the NIC, share the same I/O interconnect (i.e. the PCI bus). Another interesting fact is when having multiple instances of AES on the same device: the aggregated performance is lower compared to that of every other kernel combination on a given device, as shown in Table II. GTX 1080 Ti is an exception as it is not affected by the compute-intensive nature of AES and is able to sustain peak performance even when four AES instances are concurrently executed. On that note, despite that the integrated GPU performs tolerably well on single AES execution when combined with a large batch size (Figure 1), it is the least suitable device when the desired scenario requires the concurrent execution of an AES instance alongside an instance of any other application, as shown in the bottom part of Table II. Moreover, when DPI is coupled with MD5 (Tables I and III), the GTX 1080 Ti seems to have a twofold drawback as its performance is poor while being the most energy-hungry device of the system. The CPU and the integrated GPU both achieve similar performance results, but in the case of the UHD graphics card, top performance can be sustained regardless of the batch size. These observations lead us to a conclusion that in the presence of those two applications, by offloading the workload to the integrated graphics card we sustain top performance while keeping the latency low and we also keep the CPU and the discrete GPU idle, which either promotes the energy efficiency of the system or provides room for the execution of at least one computation-intensive application, like AES, without sacrificing performance.

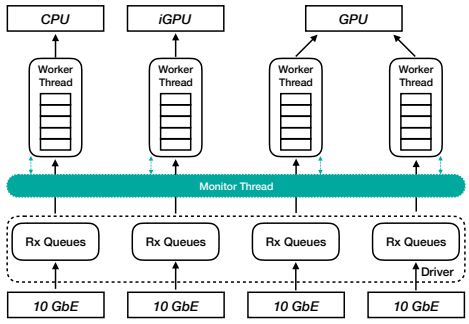

Fig. 3. Pythia architecture for processing network packets.

TABLE I
PERFORMANCE CHARACTERIZATION OF DPI.

					Performance per Kernel		Aggregated Performance
Device	**Application**	**Buffer size**	**Co-workers**	**Co-worker**	**Gbps**	**Slow-down**	**Gbps**
GTX1080Ti	DPI	1024	1	MD5	12.9	14.6%	25.8
			3		6.9	54.3%	27.6
		4096	1		12.0	33.3%	23.9
			3		6.9	61.7%	27.6
		16384	1		8.5	49.4%	**17.7**
			3		4.7	72.0%	**19.8**
		1024	1	AES	13.3	11.9%	26.7
			3		6.9	54.3%	27.6
		4096	1		12.9	28.3%	25.8
			3		6.9	61.7%	27.6
		16384	1		7.2	57.1%	**14.3**
			3		4.9	70.8%	**20.0**
		1024	1	DPI	13.8	8.6%	27.6
			3		6.9	54.3%	27.6
		4096	1		13.1	27.2%	26.1
			3		6.9	61.7%	27.6
		16384	1		8.9	47.0%	**17.9**
			3		5.3	68.5%	**21.2**
i7-8700K	DPI	1024	1	MD5	14.1	45.8%	27.1
			3		6.7	74.2%	27.5
		4096	1		14.5	42.9%	27.2
			3		6.7	73.6%	27.3
		16384	1		14.6	42.7%	25.7
			3		6.7	73.7%	26.6
		1024	1	AES	18.6	28.5%	27.2
			3		8.1	68.8%	21.8
		4096	1		16.5	35.0%	25.5
			3		7.2	71.7%	20.9
		16384	1		15.3	40.0%	24.0
			3		6.5	74.5%	20.3
		1024	1	DPI	15.0	42.3%	26.9
			3		7.1	72.7%	27.3
		4096	1		14.5	42.9%	27.2
			3		7.1	72.0%	27.1
		16384	1		14.8	41.2%	25.6
			3		6.7	73.7%	25.6
UHDGraphics	DPI	1024	1	MD5	13.4	47.7%	**26.9**
			3		6.9	73.0%	**27.4**
		4096	1		13.8	50.0%	**27.6**
			3		6.9	75.0%	**27.6**
		16384	1		13.8	50.2%	**27.7**
			3		6.9	75.1%	**27.6**
		1024	1	AES	9.4	63.3%	18.9
			3		3.8	85.2%	15.1
		4096	1		11.3	59.1%	22.6
			3		4.3	84.4%	17.2
		16384	1		11.7	57.8%	23.4
			3		4.6	83.4%	18.2
		1024	1	DPI	13.3	48.0%	26.5
			3		6.8	73.4%	27.1
		4096	1		13.8	50.0%	27.6
			3		6.9	75.0%	27.6
		16384	1		13.8	50.2%	27.6
			3		6.9	75.1%	27.6

IV. REAL-TIME SCHEDULING

Our scheduler is based on a lock-free architecture model, as illustrated in Figure 3[3]. Each worker is responsible for capturing the network traffic from a set of bound network interfaces, spawning the execution of a kernel on a target device and collecting performance metrics.

As a first phase, our scheduler uses an offline analysis tool that creates all the possible application combinations, tests them on every device combination and gathers the resulted performance statistics (as described in Section III-0b). The scheduler is using these collected results when processing the incoming network traffic, to select the optimal configuration, according to a user specified policy. In particular, we create a special worker, called *monitor*, that keeps track of the active configuration and manage how efficiently it distributes its resources to workers. The monitor executes periodically (using an `ALARM` signal) to (i) decide if the active configuration is still performing better than any other configuration, and to (ii) update the performance statistics of the current active configuration to match the most recent performance statistics of the system. By doing so, our scheduler is able to adapt to

[3]A detailed comparison of different models can be found at [8].

TABLE II
PERFORMANCE CHARACTERIZATION OF AES.

					Performance per Kernel		Aggregated Performance
Device	**Application**	**Buffer size**	**Co-workers**	**Co-worker**	**Gbps**	**Slow-down**	**Gbps**
GTX1080Ti	AES	1024	1	MD5	13.8	26.2%	27.6
			3		6.9	63.1%	27.6
		4096	1		12.1	34.2%	24.3
			3		6.9	62.5%	27.6
		16384	1		8.4	49.4%	**17.7**
			3		4.7	71.7%	**19.4**
		1024	1	AES	13.8	26.2%	27.6
			3		6.9	63.1%	27.6
		4096	1		13.2	28.3%	26.3
			3		6.9	62.5%	27.6
		16384	1		8.9	46.4%	**17.8**
			3		5.3	68.1%	**21.2**
		1024	1	DPI	13.3	28.9%	26.4
			3		6.9	63.1%	27.6
		4096	1		13.3	27.7%	26.5
			3		6.9	62.5%	27.6
		16384	1		7.4	55.4%	**14.8**
			3		4.8	71.1%	**19.4**
i7-8700K	AES	1024	1	MD5	10.3	41.8%	27.4
			3		5.1	71.2%	27.3
		4096	1		10.0	45.4%	25.8
			3		5.2	71.6%	26.6
		16384	1		9.2	49.2%	24.5
			3		5.4	70.2%	26.6
		1024	1	AES	10.0	43.5%	**18.9**
			3		4.8	72.9%	**18.3**
		4096	1		9.7	46.7%	**18.6**
			3		4.6	74.9%	**17.9**
		16384	1		9.6	47.0%	**18.4**
			3		4.6	74.6%	**17.7**
		1024	1	DPI	9.5	46.3%	27.3
			3		4.8	72.9%	26.9
		4096	1		9.1	50.3%	25.4
			3		4.8	73.8%	26.0
		16384	1		9.0	50.3%	23.1
			3		5.0	72.4%	24.7
UHDGraphics	AES	1024	1	MD5	9.6	10.3%	**19.3**
			3		6.6	38.3%	26.3
		4096	1		10.9	6.0%	**21.9**
			3		6.8	41.4%	27.0
		16384	1		11.6	7.2%	**23.2**
			3		6.8	45.6%	27.1
		1024	1	AES	5.9	44.9%	**11.8**
			3		3.1	71.0%	**12.4**
		4096	1		6.6	43.1%	**13.2**
			3		3.5	69.8%	**13.9**
		16384	1		6.9	44.8%	**13.8**
			3		3.6	71.2%	**14.5**
		1024	1	DPI	9.4	12.1%	18.8
			3		6.5	39.3%	25.9
		4096	1		11.3	2.6%	22.6
			3		6.8	41.4%	27.0
		16384	1		11.7	6.4%	23.4
			3		6.8	45.6%	27.1

TABLE III
PERFORMANCE CHARACTERIZATION OF MD5.

					Performance per Kernel		Aggregated Performance
Device	**Application**	**Buffer size**	**Co-workers**	**Co-worker**	**Gbps**	**Slow-down**	**Gbps**
GTX1080Ti	MD5	1024	1	MD5	13.8	37.0%	27.6
			3		6.9	68.5%	27.6
		4096	1		13.6	32.3%	27.3
			3		6.9	65.7%	27.6
		16384	1		9.1	48.9%	**18.2**
			3		5.3	70.2%	**21.3**
		1024	1	AES	13.8	37.0%	27.6
			3		6.9	68.5%	27.6
		4096	1		12.2	39.3%	24.3
			3		6.9	65.7%	27.6
		16384	1		8.9	50.0%	**17.6**
			3		5.1	71.3%	**19.3**
		1024	1	DPI	12.7	42.0%	25.5
			3		6.9	68.5%	27.6
		4096	1		12.0	40.3%	23.9
			3		6.9	65.7%	27.6
		16384	1		9.2	48.3%	**17.8**
			3		4.9	72.5%	**18.5**
i7-8700K	MD5	1024	1	MD5	14.1	47.2%	27.4
			3		7.0	73.8%	27.5
		4096	1		13.8	49.5%	27.4
			3		7.2	73.6%	27.4
		16384	1		13.9	42.8%	27.1
			3		7.2	70.4%	27.1
		1024	1	AES	17.8	33.3%	27.2
			3		8.9	66.7%	22.9
		4096	1		16.2	40.7%	26.0
			3		7.8	71.4%	21.8
		16384	1		15.1	37.9%	24.3
			3		7.1	70.8%	21.3
		1024	1	DPI	13.8	48.3%	27.1
			3		7.0	73.8%	27.4
		4096	1		14.5	46.9%	27.2
			3		6.4	76.6%	27.1
		16384	1		13.3	45.3%	25.5
			3		6.8	72.0%	26.3
UHDGraphics	MD5	1024	1	MD5	13.6	49.3%	27.2
			3		6.9	74.3%	27.4
		4096	1		13.8	50.2%	27.6
			3		6.9	75.1%	27.6
		16384	1		13.8	50.2%	27.6
			3		6.9	75.1%	27.6
		1024	1	AES	9.7	63.8%	19.4
			3		3.8	85.8%	15.4
		4096	1		11.2	59.6%	22.5
			3		4.3	84.5%	17.1
		16384	1		11.6	58.1%	23.2
			3		4.6	83.4%	18.1
		1024	1	DPI	13.5	49.6%	**27.1**
			3		6.8	74.6%	**27.1**
		4096	1		13.8	50.2%	**27.6**
			3		6.9	75.1%	**27.6**
		16384	1		13.8	50.2%	**27.7**
			3		6.9	75.1%	**27.6**

traffic rate changes quickly and also re-train itself over time. A representative set of policies that we have implemented so far, include (a) *throughput maximization* in which we seek to optimize in terms of aggregated processing rate (typically at the cost of increased latency and power consumption; (b) *latency minimization*, that can be applied to real-time or latency-critical applications; and (c) *energy consumption minimization*.

V. EVALUATION

Figures 4(a)-(d) show the performance of our scheduler for a representative fraction of applications (i.e., AES and DPI) when (i) fluctuating network traffic rate and (ii) changing policies on-the-fly. For the former, we use a policy to handle all input traffic at highest energy efficiency. The traffic rate is low enough for a single device to cope with it, which results to significantly low power consumption. This is not the case in the second experiment, in which we seek the maximum possible throughput before aggressively switching to an energy-efficient policy. For comparison, we also display the maximum power consumption when both devices are exhaustively used simultaneously. The observed variability in latency is the result of the dynamic scheduler decisions regarding the batching and device selection. Overall, our scheduler is capable to adapt to a highly diverse computational demand among different applications, producing live decisions that aim to maintain the maximum energy efficiency and to avoid excessive latency (besides the requested performance policy).

a) Throughput: As shown in Figures 4(c) and 4(d), our system is able to process a constant traffic rate of almost 20 Gbps when a single applications is active and at about 30 Gbps in the scenario of two active applications (0-15 seconds mark). When the traffic rate varies, our scheduling schema manages to cope with up to 10 Gbps input traffic rate per application as shown in Figures 4(a) and 4(b) (0-20 seconds mark). When the traffic rate changes, such as the increase from 20 to 40 Gbps (Figure 4(b)), a second device (in this case the GTX 1080 Ti) is enabled to increase the computational capacity of the system. An interesting time interval exists between the 20th and the 30th second mark of Figure 4(a) when the discrete GPU is activated but is immediately deactivated, as the monitoring reveals that only the presence of the integrated graphics card can still cope with the incoming traffic. The GTX 1080 Ti is only re-activated when the traffic rate is doubled to 40 Gbps.

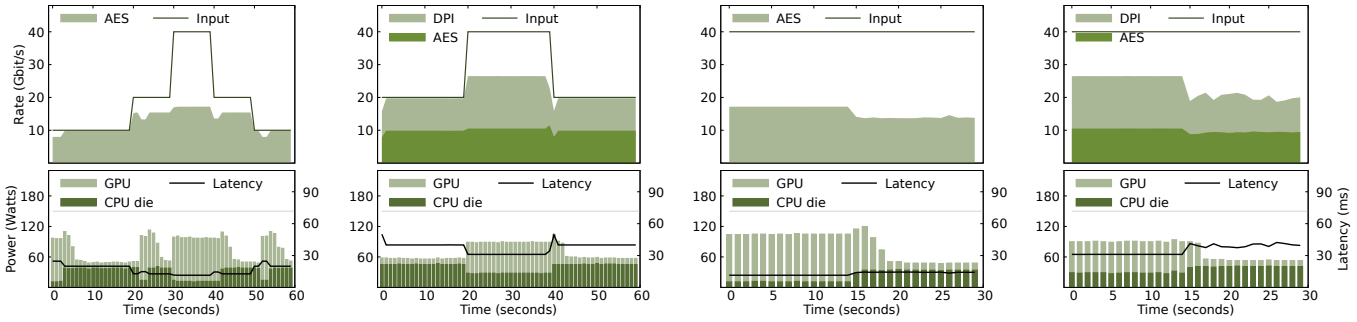

Fig. 4. Adaptive scheduling for different workload combinations under different conditions: network traffic rate fluctuations (a)-(b), and policy changes (c)-(d).

TABLE IV
COMPARISON WITH STATE-OF-THE-ART TOOLS.

	Target architecture		Concurrent app support	
Related work	Single Dev	**Multi Devs**	Single App	**Multi Apps**
GASPP [3] *(comp. offloading)*	✓	–	✓	–
APUnet [4] *(comp. offloading)*	✓	–	✓	–
Dobrescu et al. [12]	✓	–	✓	✓
Papadogiannaki et al. [8]	✓	✓	✓	–
Pythia	✓	✓	✓	✓

b) Energy Efficiency: The activation of the minimum necessary number of devices, typically leads to lower power consumption. When activating a lowest power consumption policy in Figures 4(c)-(d) (15-second mark), the GTX 1080 Ti is forced to shut-down, resulting to significant energy savings. It is also clear from the results of variable traffic rate experiments (Figures 4(a) and 4(b)) that only when an increase in the traffic rate occurs and more computational capacity is needed, the system activates an extra device at the cost of greater energy expenditure.

c) Latency: An increase of the batch size usually results to higher throughput, but also to increased latency. Overall, we try to minimize latency up to a point where no interference with the requested policy occurs. For example, even when the goal is to maximize the overall throughput of the system (Figure 4(b)), during the second 20-seconds interval, latency remains considerably low despite the fact that the discrete GPU is active as the traffic characteristics demands so. The reason behind this is not only the presence of an extra device, but mainly because the system does recognize that an even larger batch size would not result in extra performance gains.

VI. Related Work

A comparison of Pythia to the most relevant state-of-the-art tools is shown in Table IV. GASPP [3] shows an extreme approach that delivers all packets directly to a high-end GPU for processing, while APUNet [4] utilizes GPUs that are integrated in the CPU die, to alleviate the overhead of extra memory copies. Papadogiannaki et al [8] propose an adaptive scheduling approach that uses performance policies to determine the appropriate combination of devices for efficient execution of network packet processing applications. In this work, we extend this solution by enabling the multiplexing of different network functions across heterogeneous devices. Finally, there is ongoing work on providing performance predictability [12] and fair queuing [13] when running a diverse set of applications that contend for shared resources.

VII. Conclusions

We proposed an adaptive scheduling solution that enables real-time application multiplexing across heterogeneous processors, and is able to respond quickly to network fluctuations or system changes. As part of our future work, we plan to optimize the complexity of the offline analysis phase by taking into consideration, a-priori, the specific characteristics of the available processing devices.

Acknowledgements

This work was supported by the projects CONCORDIA, I-BiDaaS and C4IIoT, funded by the European Commission under Grant Agreements No. 830927, No. 780787 and No.833828. This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which maybe made of the information contained therein.

References

[1] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High Performance Network Intrusion Detection Using Graphics Processors," in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, 2008.

[2] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "MIDeA: A Multi-Parallel Intrusion Detection Architecture," in *CCS*, 2011.

[3] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "GASPP: A GPU-Accelerated Stateful Packet Processing Framework," in *USENIX ATC*, 2014.

[4] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park, "APUNet: Revitalizing GPU as Packet Processing Accelerator," in *USENIX NSDI*, 2017.

[5] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a single compute device image in OpenCL for multiple GPUs," in *PPoPP*, 2011.

[6] G. Maier, A. Feldmann, V. Paxson, and M. Allman, "On dominant characteristics of residential broadband internet traffic," in *IMC*, 2009.

[7] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding Data Center Traffic Characteristics," *SIGCOMM CCR*, vol. 40, no. 1, January 2010.

[8] E. Papadogiannaki, L. Koromilas, G. Vasiliadis, and S. Ioannidis, "Efficient software packet processing on heterogeneous and asymmetric hardware architectures," *IEEE/ACM Transactions on Networking*, vol. 25, no. 3, pp. 1593–1606, 2017.

[9] "The Snort IDS/IPS," Available: http://www.snort.org/.

[10] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker, "Packet caches on routers: the implications of universal redundant traffic elimination," in *ACM SIGCOMM*, 2008.

[11] L. Rizzo, "netmap: A Novel Framework for Fast Packet I/O," in *USENIX ATC*, 2012.

[12] M. Dobrescu, K. Argyraki, and S. Ratnasamy, "Toward Predictable Performance in Software Packet-Processing Platforms," in *USENIX NSDI*, 2012.

[13] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica, "Multi-Resource Fair Queueing for Packet Processing," in *ACM SIGCOMM*, 2012.