Efficient Software Packet Processing on Heterogeneous and Asymmetric Hardware Architectures

Lazaros Koromilas
FORTH
koromil@ics.forth.gr

Giorgos Vasiliadis
FORTH
gvasil@ics.forth.gr

Ioannis Manousakis
Rutgers University
ioannis.manousakis
@cs.rutgers.edu

Sotiris Ioannidis
FORTH
sotiris@ics.forth.gr

ABSTRACT

Heterogeneous and asymmetric computing systems are composed by a set of different processing units, each with its own unique performance and energy characteristics. Still, the majority of current network packet processing frameworks targets only a single device (the CPU or some accelerator), leaving other processing resources idle. In this paper, we propose an adaptive scheduling approach that supports heterogeneous and asymmetric hardware, tailored for network packet processing applications. Our scheduler is able to respond quickly to dynamic performance fluctuations that occur at real-time, such as traffic bursts, application overloads and system changes. The experimental results show that our system is able to match the peak throughput of a diverse set of packet processing workloads, while consuming up to 3.5x less energy.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*heterogeneous (hybrid) systems*

Keywords

Packet processing; packet scheduling; OpenCL; heterogeneous processing

1. INTRODUCTION

The advent of commodity heterogeneous systems (i.e. systems that utilize multiple processor types, typically CPUs and GPUs) has motivated network developers and researchers to exploit alternative architectures, and utilize them in order to build high-performance and scalable network packet processing systems [19, 22, 23, 32, 40], as well as systems optimized for lower power envelop [29]. Unfortunately, the majority of these approaches often target a single computational *device* [1], such as the multicore CPU or a powerful GPU, leaving other devices idle. Developing an application that can utilize *each* and *every* available device effectively and consistently, across a wide range of diverse applications, is highly challenging.

Heterogeneous, multi-device systems typically offer system designers different optimization opportunities that offer inherent trade-offs between energy consumption and various performance metrics — in our case, forwarding rate and latency. The challenge to fully tap a heterogeneous system, is to effectively map computations to processing devices, and do so as automated as possible. Previous work attempted to solve this problem by developing load-balancing frameworks that automatically partition the workload across the devices [13, 24, 27]. These approaches either assume that all devices provide equal performance [24] or require a series of small execution trials to determine their relative performance [13, 27]. The disadvantage of such approaches is that they have been designed for applications that take as input constant streaming data, and as a consequence, they are slow to adapt when the input data stream varies. This makes them extremely difficult to apply to network infrastructure, where traffic variability [12, 28] and overloads [14] significantly affect the utilization and performance of network applications.

In this paper, we propose an adaptive scheduling approach that exploits highly heterogeneous systems and is tailored for network packet processing applications. Our proposed scheduler is designed to explicitly account for the heterogeneity of *(i)* the hardware, *(ii)* the applications and *(iii)* the incoming traffic. Moreover, the scheduler is able to quickly respond to dynamic performance fluctuations that occur at run-time, such as traffic bursts, application overloads and system changes.

The contributions of our work are:

- We characterize the performance and power consumption of several representative network applications on

[1]Hereafter, we use the term device to refer to computational devices, such as CPUs and GPUs, unless explicitly stated otherwise.

ANCS'14, October 20–21, 2014, Los Angeles, CA, USA.

http://dx.doi.org/10.1145/2658260.2658265.

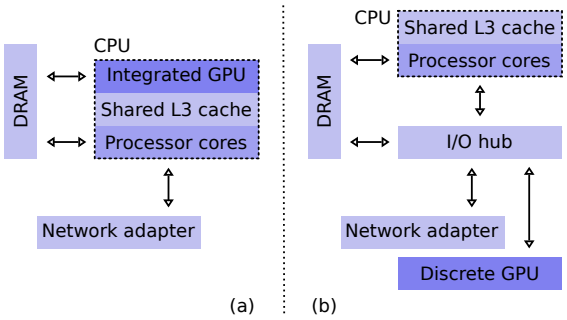


Figure 1: Architectural comparison of packet processing on an (a) integrated and (b) discrete GPU.

heterogeneous, commodity multi-device systems. We show how to combine different devices (i.e. CPUs, integrated GPUs and discrete GPUs) and quantify the problems that arise by their concurrent utilization.

- We show that the performance ranking of different devices has wide variations when executing different classes of network applications. In some cases, a device can be the best fit for one application, and, at the same time, the worst for another.
- Motivated by the previous deficiency, we propose a scheduling algorithm that, given a single application, effectively utilizes the most efficient device (or group of devices) based on the current conditions. Our proposed scheduler is able to respond to dynamic performance fluctuations —such as traffic bursts, application overloads and system changes— and provide consistently good performance.

2. BACKGROUND

Typical commodity hardware architectures offer heterogeneity at three levels: *(i)* at the traditional x86 CPU architecture, *(ii)* at an integrated GPU packed on the same processor die, and *(iii)* at a discrete high-end GPU. All three devices have unique performance and energy characteristics. Overall, the CPU cores are good at handling branch-intensive packet processing workloads, while discrete GPUs tend to operate efficiently in data-parallel workloads. Between those two is the integrated GPU which features high energy efficiency without significantly compromising the processing rate or latency. Typically, the discrete GPU and the CPU communicate over the PCIe bus and they do not share the same physical address space (although this might change in the near future). The integrated GPU on the other hand, shares the LLC cache and the memory controller of the CPU.

2.1 Architectural Comparison

In Figure 1 (right side), we illustrate the packet processing scheme that has been used by approaches that utilize a discrete GPU [22, 37–40]. The majority of these approaches perform a total of seven steps (assuming that a packet batch is already in the NICs internal queue): the DMA transaction between the network interface and the main memory, the transfer of the packets to the I/O region which corresponds to the discrete GPU (this operation traditionally invokes CPU caches, but the cache pollution can be minimized by using *non-temporal* data move instructions) the DMA transaction towards the memory space of the GPU, the actual computational GPU kernel itself and the transfer of the results back to the host memory. All data transfers must operate on fairly large chunks of data, due to the PCIe interconnect inability to handle small data transfers efficiently. The equivalent architecture, using an integrated GPU that is packed on the CPU die, is illustrated on the left side of Figure 1. The advantage of this approach is that the integrated GPU and CPU share the same physical memory address space, which allows in-place data processing. This results to fewer data transfers and hence lower processing latency. This scheme also has lower power consumption, as the absence of the I/O Hub alone saves 20W of energy, when compared to the discrete GPU setup of Figure 1(b).

2.2 Quantitative Comparison

The integrated GPU (such as the HD Graphics 4000 we used in this work) has higher energy efficiency as a computational device, compared to modern processors and GPUs. The reason is threefold. First, integrated GPUs are typically implemented with low-power, 3D transistor manufacturing process. Second, they have a simple internal architecture and no dedicated main memory. Third, they match the computational requirements of applications, in which the main bottleneck is the I/O interface and thus, a discrete GPU would be under-utilized. In Section 3.3.2 we show, in more detail, the energy efficiency of these devices when executing typical network packet processing workloads.

3. SYSTEM SETUP

We will now describe the hardware setup, and our power instrumentation and measurement scheme. Our scheme is capable of accurately measuring the power consumption of various hardware components, such as the CPU and GPU, in real time. We also describe the packet processing applications that we used for this work and show how we parallelized them using OpenCL, to efficiently execute in each of the three processing devices.

3.1 Hardware Platform

Our base system is equipped with one Intel Core i7-3770 Ivy Bridge processor and one NVIDIA GeForce GTX 480 graphics card. The processor contains four CPU cores operating at 3.4GHz, with hyper-threading support, resulting in eight hardware threads, and an integrated HD Graphics 4000 GPU. Overall, our system contains *three* different, heterogeneous, computational devices: one CPU, one integrated GPU and one discrete GPU. The system is equipped with 8GB of dual-channel DDR3-1333 DRAM with 25.6 GB/s throughput. The L3 cache (8MB) and the memory controller are shared across the CPU cores and the integrated GPU. Each CPU core is equipped with 32KB of L1 cache and 256KB of L2 cache. The GTX 480 has 480 cores in 15 multiprocessors and 1280 MB of GDDR5 memory. The HD Graphics 4000 has 16 execution units, a 64-hardware thread dispatcher and a 100 KB texture cache. The maximum estimated performance of this GPU is rated at 294 GFlop/s on the maximum operating frequency of 1150 Mhz [7]. While Intel does not provide it's Thermal Design Power (TDP) limit, we estimate that it is close to 17 Watt. For the whole processor die the TDP is 77 Watt.

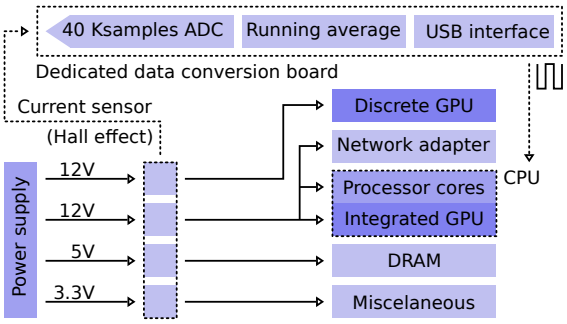


Figure 2: Our power instrumentation scheme. We use four current sensors to monitor (real-time) the consumption of the CPU, GPU, DRAM and miscellaneous motherboard peripherals.

We notice that our hardware platform exposes an interesting design trade-off: even though the on-chip GPU has fewer resources (i.e. hardware threads, execution units, register file) than a high-end discrete graphics card, it is directly connected to the CPU and the main memory via a fast on-chip ring bus, and has much lower power consumption. As we will see in Section 3.3.2, this design is well suited for applications in which the overall performance is limited by the I/O subsystem, and not by the computational capacity.

3.1.1 Power Instrumentation

To accurately measure the power consumption of our hardware system, we have designed the hardware instrumentation scheme shown in Figure 2. Our scheme is capable of high-rate, 1 KHz measurement, and also provides a breakdown of system consumption into four distinct components: Processor, Memory, Network and Discrete GPU.

Specifically, we utilize four high-precision *Hall effect* current sensors to constantly monitor the three ATX power-supply power lines (+12.0a, +12.0b +5.0, +3.3 Volts). The sensors [2], coupled with the interface kit [1], costs less than 110$. The analog sensor values are converted into digital values, and transmitted over a USB interface to a dedicated data logger board. The data logger includes a high-speed analog-to-digital converter (ADC) operating at a frequency of 40 KHz. The output data produced by the ADC are continuously read by a custom firmware running on the board, which also applies a running-average filter and periodically interrupts the processor with a rate of 1 KHz to report the values. A daemon, running in our base server, periodically collects the measurements from the data logger, and makes them available for monitoring and control. We take advantage of the physical layout to achieve a breakdown of the total power consumption: the 12Va line powers the processor, the 12Vb powers the GPU, the 5V line powers the memory, and the 3.3V line powers the rest of the peripherals on the motherboard. The 12Va line also feeds the 10GbE NICs. To calculate their power consumption, we use a utilization-based model.

3.2 Workloads

We implemented four typical packet processing applications, using OpenCL [3], that are typically deployed in network appliances and involve both computational and memory-intensive behavior.

3.2.1 IPv4 Packet Forwarding

An IP packet forwarder is one of the simplest packet processing applications. Its main function is the reception and transmission of network packets from one network interface to another. Before packet transmission, the forwarder checks the integrity of the IP header, drops corrupted packets and rewrites the destination IP address according to the specified configuration. Other functions include decrementing the Time To Live (TTL) field. If the TTL field of an incoming packet is zero, the packet is dropped and an ICMP Time Exceeded message is transmitted to the sender. We implemented the RadixTrie lookup algorithm and used a routing table of 17,000 entries.

3.2.2 Deep Packet Inspection

Deep packet inspection (DPI) is a common operation in network traffic processing applications. It is typically used in traffic classification and shaping tools, as well as in network intrusion detection and prevention systems. We ported a DFA implementation of the Aho-Corasick algorithm [10] for string searching, and used the `content` patterns (about 10,000 fixed strings) of the latest Snort [5] distribution, which we compiled into the same state machine.

3.2.3 Packet Hashing

Packet hashing is used in redundancy elimination and in-network caching systems [9, 11]. Redundancy elimination systems typically maintain a "packet store" and a "fingerprint table" (that maps content fingerprints to packet-store entries). On reception of a new packet, the packet store is updated, and the fingerprint table is checked to determine whether the packet includes a significant fraction of content cached in the packet store; if yes, an encoded version that eliminates this (recently observed) content is transmitted. We have implemented the MD5 algorithm [31], since it has low probability of collisions and is also commonly used for checking data integrity [4] and deduplication [25].

3.2.4 Encryption

Encryption is used by protocols and services, such as SSL, VPN and IPsec, for securing communications by authenticating and encrypting the IP packets of a communication session. We implemented AES-CBC encryption using a different 128-bit key for each connection. This is a representative form of computational-intensive packet processing.

3.3 Packet-processing Parallelization

To execute the packet processing applications uniformly across the different devices of our base system, we implemented them on top of OpenCL 1.1. Our aim was to develop a portable implementation of each application, that can also run efficiently on each device. We used the Intel OpenCL runtime 2.0 for the Core processor family, Intel HD Graphics driver 9.17, as well as the OpenCL implementation that comes with NVIDIA CUDA Toolkit 5.0. Due to space constraints we omit the full details of our implementation, and we only list the most important design aspects and optimizations that we addressed.

Each of our representative applications, is implemented as a different compute kernel. In OpenCL, an instance of a compute kernel is called a *work-item*; multiple work-items are grouped together and form *work-groups*. We followed a thread-per-packet approach, similar to [17, 19, 40], and as-

signed each work-item to handle (at least) one packet; each work-item reads the packet from the device memory and performs the processing. As different work-groups can be scheduled to run concurrently on different hardware cores, the choice of work-groups number provides an interesting trade-off: a large number of work-groups provides more flexibility in scheduling, but also increases the switching overhead. GPUs contain a significantly faster thread scheduler, thus it is better to spawn a large number of work-groups to hide memory latencies: while a group of threads waits for data fetching, another group can be scheduled for execution. CPUs, on the other hand, perform better when the number of different work-groups is equal to the number of the underlying hardware cores.

When executing compute kernels on the discrete GPU, the first thing to consider is how to transfer the packets to and from the device. Discrete GPUs have a memory space that is physically independent from the host. To execute a task, explicit data transfers between the host and the device are required. The transfers are performed via DMA, hence the host memory region should be page-locked to prevent page swapping during the transfers. Additionally, a data buffer required for the execution of a computing kernel has to be created and associated to a specific *context*; devices from different *platforms* (i.e. heterogeneous) cannot belong to the same context in the general case, and thus, cannot share data directly.[2] To overcome this, we explicitly copied received packets to a separate, page-locked, buffer that has been allocated from the discrete GPU's context and can be transferred safely via DMA. The data transfers and the GPU execution are performed asynchronously, to allow overlap of computation and communication and further improve parallelism. Whenever a batch of packets is transferred and/or processed by the GPU, newly arriving packets are copied to another batch in a pipeline fashion. We notice that different applications require different data transfers across the discrete GPU. For instance, DPI and MD5 do not alter the contents of the packets, hence it is not needed to transfer them back; they are already stored in the host memory. Packets have to be transferred back, when processed by the AES and the IP Forwarder applications, as both applications alter their contents. Still, the IP Packet Forwarder processes and modifies only the packet headers. In order to prevent redundant data transfers, we only transfer the headers of each packet to and from the GTX 480, for the IP Packet Forwarder case; the packet headers are stored separately in sequential header descriptors (128 bytes each), a technique already supported by modern NICs [6]. Nevertheless, these data transfers are unnecessary when the processing is performed by the CPU or the integrated GPU, as both devices have direct access to the host memory. To avoid the extra copies, we explicitly mapped —using the `clEnqueueMapBuffer()` function— the corresponding memory buffers directly to the CPU and the integrated GPU.

Accessing data in the global memory is critical to the performance of all of our representative applications. GPUs require column-major order to enable memory loads to be more effective, the so-called memory coalescing [30]. CPUs require row-major order to preserve the cache locality within each thread. As the impacts of the two patterns are contradictory, we first tried to transpose the whole packets in GPU memory only, and benefit from memory coalescing. The overall costs, however, pay off only when accessing the memory with small vector types (i.e. `char4`); when using the `int4` type though, the overhead was not amortized by the resulting memory coalescing gains, in none of our representative applications. Besides the GPU gains, the CPU enables the use of SIMD units when using the `int4` type, because the vectorized code is translated to SIMD instructions [33]. To that end, we redesigned the input process and access the packets using `int4` vector types in a row-major order, for both the CPU and the GPU.

Finally, OpenCL provides a memory region, called *local memory*, that is shared by all work-items of a work-group. The local memory is implemented as an on-chip memory on GPUs, which is much faster than the off-chip global memory. As such, GPUs take advantage of local memory to improve performance. By contrast, CPUs do not have a special physical memory designed as local memory. As a result, all memory objects in local memory are mapped into sections of global memory, and will have a negative impact on performance. To overcome this, we explicitly stage data to local memory only when performing computations on the discrete GPU.

3.3.1 Batch Processing

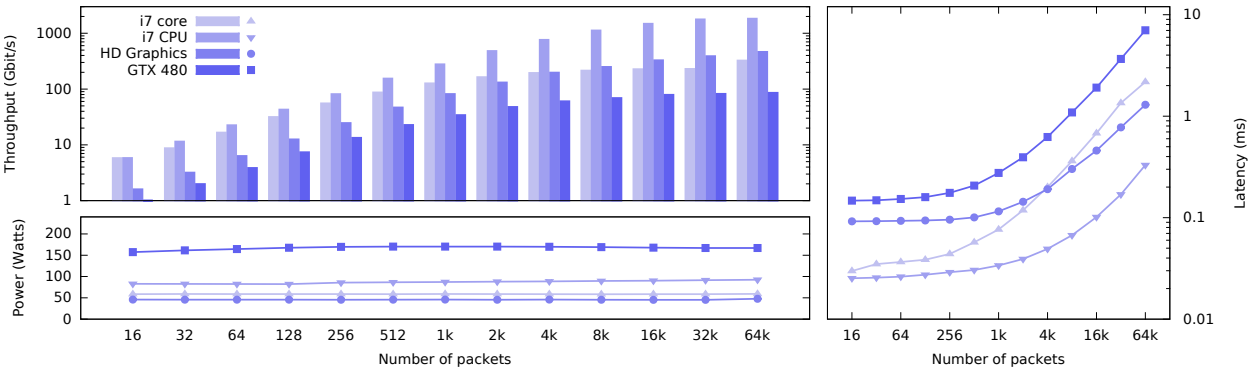
Network packets are placed into batches in the same order they are received. In case two (or more) devices are used simultaneously though, it is possible to be reordered. One solution to prevent packet reordering is to synchronize the devices using a barrier. By doing so, we enforce all the involved devices to execute in a lockstep fashion. Unfortunately, this would reduce the overall performance, as the fast devices will always wait for the slow ones. This can be a major disadvantage in setups where the devices have large computational capacity discrepancies. To overcome this limitation, we first classify incoming packets to flows before creating the batches (by hashing the 5-tuple of each packet), and ensure that packets that belong to the same flow will never be placed in batches that will execute simultaneously to different devices.

Batches are delivered to the corresponding devices, by the CPU core that is responsible to handle the traffic of each network interface. Each device has a different queue —that is allocated within the device's context— where newly arrived batches of packets are inserted.

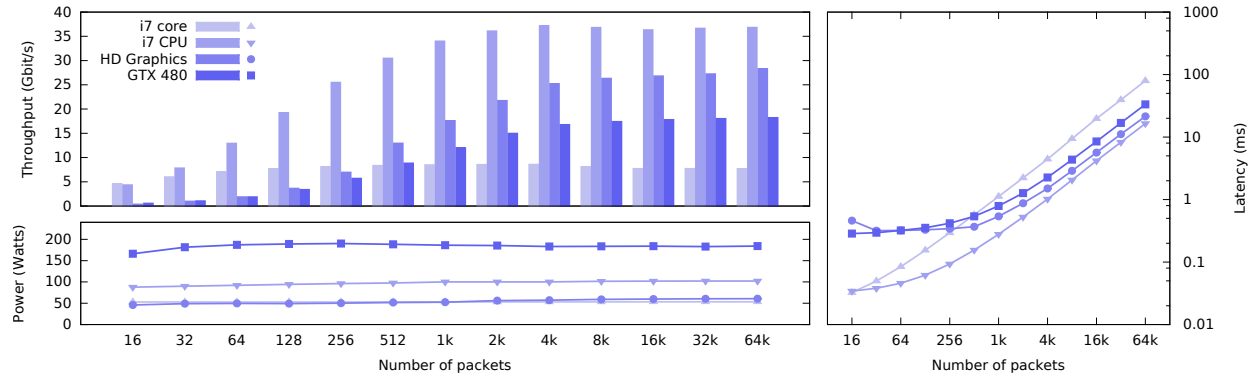
3.3.2 Performance Characterization

In this section we present the performance achieved by our applications. Specifically, we measure the sustained throughput, latency and power consumption for each of the devices that are available in our base system. We use a synthetic packet trace and a different packet batch size each time. To accurately measure the power spent for each device to process the corresponding batch, we measure the power consumption of *all* the components that are required for the execution. For instance, when we use the GPU for packet processing, the CPU has to collect the necessary packets, transfer them to the device (via DMA), spawn a GPU kernel execution, and transfer the results back to the main memory. Instead, when we use the CPU (or the integrated GPU), we power-off the discrete GPU, as it is not needed. By measuring the power consumption of the right components each time, we can accurately compare different devices.

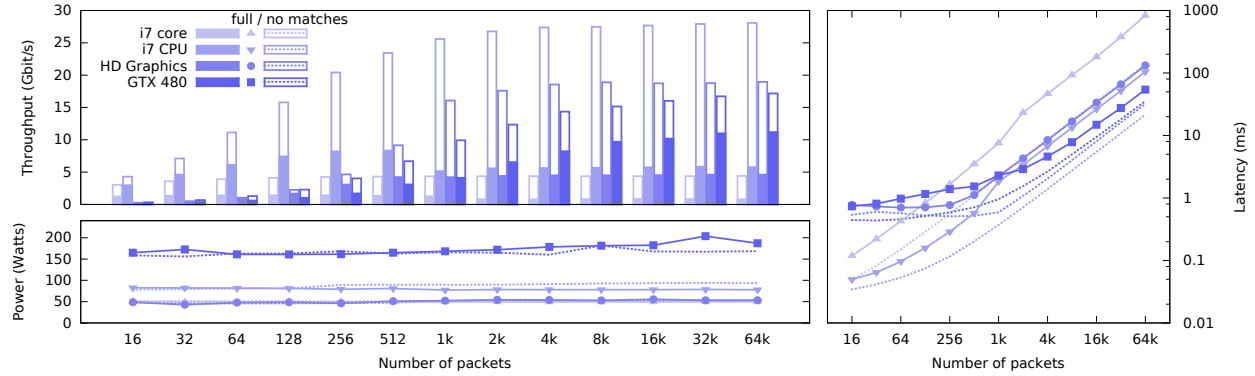
[2]Context sharing is available in the Intel OpenCL implementation [8], but this does not include our discrete GPU.



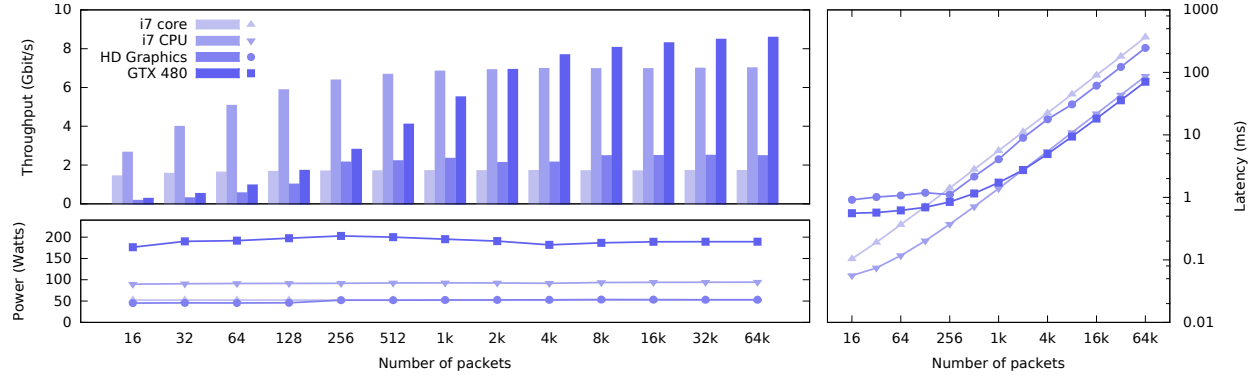
(a) IPv4 Packet Forwarding



(b) MD5

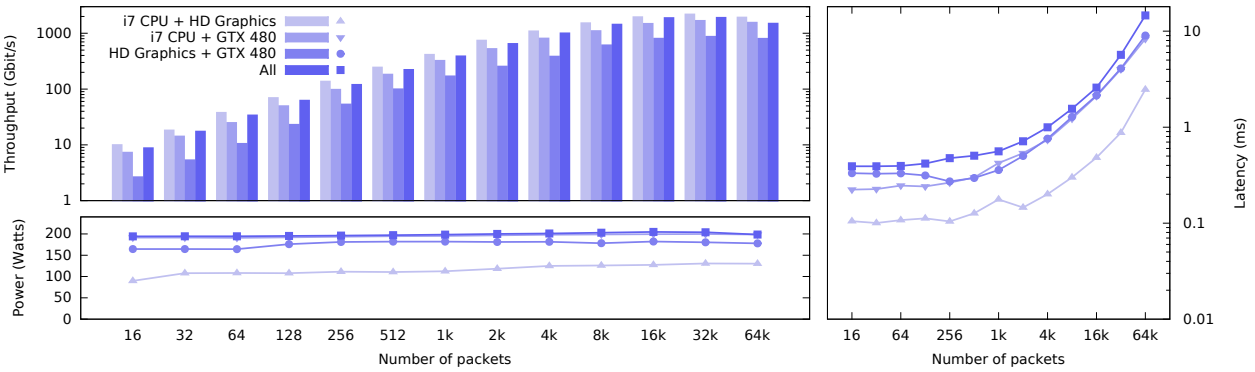


(c) DPI

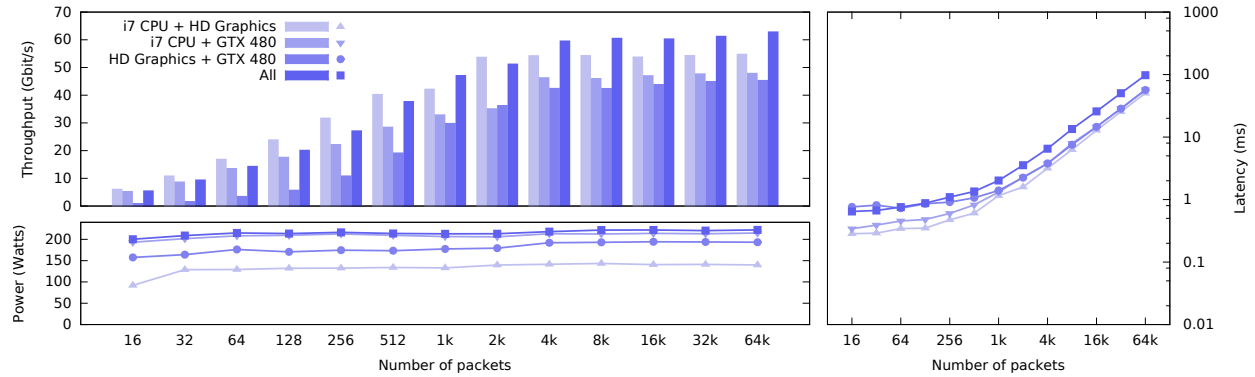


(d) AES-CBC

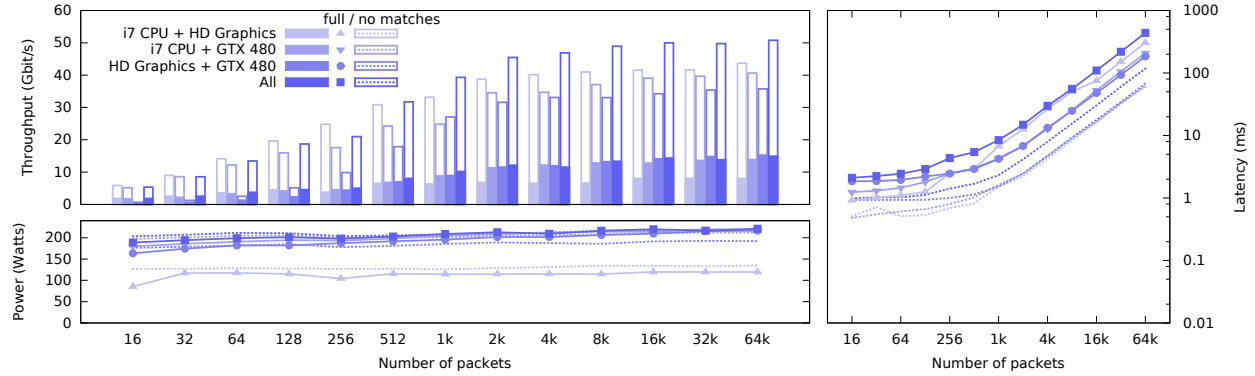
Figure 3: Throughput, latency and power consumption for (a) IPv4 packet forwarding, (b) MD5 hashing, (c) Deep Packet Inspection, and (d) AES-CBC 128-bit encryption.



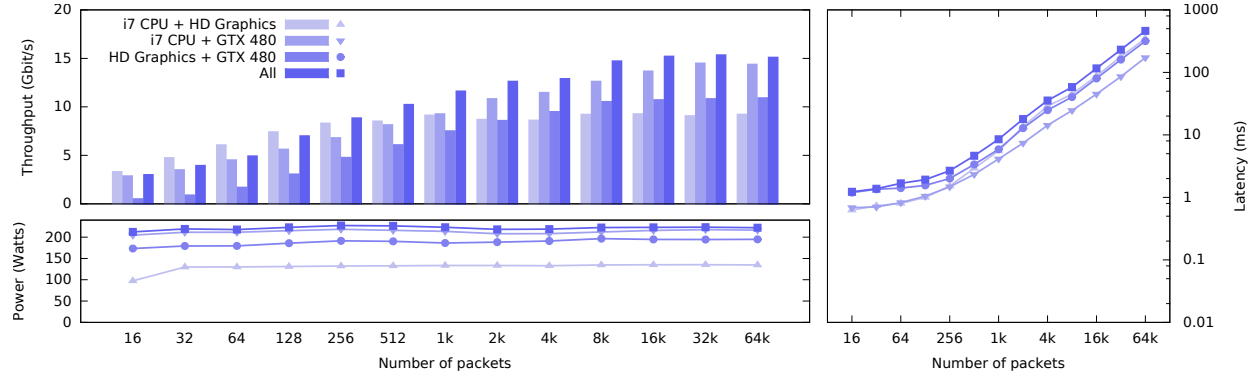
(a) IPv4 Packet Forwarding



(b) MD5



(c) DPI



(d) AES-CBC

Figure 4: Throughput, latency and power consumption for (a) IPv4 packet forwarding, (b) MD5 hashing, (c) Deep Packet Inspection, and (d) AES-CBC 128-bit encryption, for combinations of devices.

Figures 3 and 4 summarize the characteristics of each of these types of packet processing during a "solo" run (one device runs the packet-processing workload, while all the other devices are idle) and a "combo" run (more devices contribute to the packet-processing) respectively. In the combo run, the batch of packets needs to be further split into sub-batches, of different size, that will be offloaded to the corresponding devices. We have exhaustively benchmarked all possible combinations of sub-batches for each packet batch and pair of devices. Due to space constraints though, we plot only the best achieved performance for each case. In the case of the i7 processor, we include the results when using a single core only ("i7 core"), as well as all four cores in parallel ("i7 CPU"). Note that in the IPv4 forwarding application, the reported throughput corresponds to the size of full packet data, even though only their headers are processed in separate header buffers, as we described in Section 3.3.

We observe that the throughput always improves when we increase the batch size. However, different applications (as well as the same application on different devices) require a different batch size to reach their maximum throughput. Memory intensive applications (such as the IPv4 router) benefit more from large batch sizes, while computationally intensive applications (i.e. AES) require smaller batch sizes to reach the peak throughput. This is mainly the effect of cache sizes in the memory hierarchy of the specific device. For example, for the DPI in Figure 3(c) we see that a working set larger than 512 packets results in lower overall throughput for the CPU. Increasing the batch size, after the maximum throughput has been reached, results to linear increases in latency (as expected). Furthermore, we can see that the sustained throughput is not consistent across different devices. For example, the discrete GPU seems to be a good choice when performing DPI and AES on large batch of packets. The integrated GPU provides the most energy-efficient solution for all applications (even when using a single CPU-core only), however it cannot surpass the throughput of other devices (except in the case of DPI where it exceeds the discrete GPU's throughput for match-free traffic). The CPU is the best option for latency-aware setups, as it can sustain processing times below 0.1 ms for all applications. In general though, there is not a clear ranking between the devices, not even a single winner. As a matter of fact, some devices can be the best fit for some applications, and at the same time the worst option for another (as observed in the case of AES and IPv4 forwarding when executing on the GTX 480). Besides, we can see that the traffic characteristics can affect the performance of an application significantly. As we can see in Figure 3(c), the performance of DPI has large fluctuations; when there is no match in the input traffic the throughput achieved by all devices is much higher (even four to five times for the CPU) over the case where the matches overwhelm the traffic. The reason behind this is that as the number of pattern matches decreases, the DFA algorithm needs to access only a few different states. These states are stored in the cache memory, hence the overall throughput increases due to the increased cache hit ratio.

When pairing different compute devices, the resulting performance does not yield the aggregate throughput of the individual devices. For example, when executing MD5, the CPU yields 36 Gbit/s and the integrated GPU yields 28 Gbit/s, while when paired together they achieve only 54 Gbit/s. The reason behind this deviation is two-fold. First, when using devices that are packed in the same processor (i.e. the CPU and the integrated GPU), their computational capacity is capped by the internal power control unit, as they exceed the thermal constraints (TDP) of the processor. Second, they encounter resource interference, as they share the same last level cache. Actually, this is the case for all pair of devices, except in the IP Forwarding case, where the CPU alone reaches the physical limits of the memory bandwidth, hence any extra device does not help to increase the overall throughput. When using all three devices, we can see that the overall throughput is always lesser (between 17% and 22%) than the throughput of the individual devices, as a result of high memory congestion.

3.3.3 *Energy efficiency*

Figure 5 shows the energy efficiency of each packet processing application, on each computational device. The lines show the Joules that are needed to process one Mbit of data (the lower the better), under different batch size configurations (x-axis). We observe that IPv4 forwarding ends up (for the larger batch sizes) to be the most efficient application when using the i7 CPU or the integrated HD Graphics GPU, and at the same time the worst when utilizing the GTX 480 discrete GPU. We also show the efficiency of forcing the system to use only one of the four i7 cores for comparison purposes. MD5 follows the same pattern, with the gap between the integrated and discrete GPU closing in. For the case of DPI (Deep Packet Inspection) and AES encryption in CBC mode, we can see that all devices converge to about the same efficiency; such large batch sizes, however, negatively affect latency and may be impractical to use for certain scenarios. Smaller batch sizes almost always have a clear winner.

In Figure 6 we see how different combinations of devices perform with respect to energy efficiency. Compared to single-device configurations, the combinations always perform worse, even though they deliver higher aggregate throughput. Among all, the least efficient device combination is the pair of the two GPUs (HD Graphics and GTX 480), especially for small batch sizes where they remain under-utilized. On the other hand, the most efficient combination is the i7 CPU paired with its integrated HD Graphics, which deliver both low consumption and acceptable throughput.

4. EFFICIENCY VIA SCHEDULING

The performance characterization in Figures 3 and 4 indicates that there is not a clear ranking between the benchmarked computational devices. As a consequence of their architectural characteristics, some devices perform better under different metrics, while these metrics may also deviate significantly among different applications. As an example, the GTX 480 achieves the best performance for the AES encryption but the worst performance for the IP forwarding. Additionally, the traffic characteristics can affect the performance achieved by a device. For example, DPI achieves the best performance (28 Gbit/s) on the i7 CPU while there are no matches on the input traffic. On the contrary, the rate falls significantly (at 6 Gbit/s), when the matches overwhelm the traffic (which is half of the performance sustained by a GTX 480).

With these observations in mind, we propose an online scheduler that increases the efficiency of packet processing on such highly heterogeneous systems. Our scheduler ex-

plores the parameter space and selects a subset of the available computational devices to handle the incoming traffic for a given kernel. The goal of our method is to minimize: *(i)* energy consumption, and *(ii)* latency, or maximize throughput. Our scheduler consists of two phases. The first phase performs an initial, coarse profiling of each new application. In this phase, the scheduler learns the performance, latency and energy response of each device, in respect to the packet batching as well as the partitioning of each batch on every device. In the second phase, the scheduler decides the best combination of available devices that meet the desired target (e.g. maximize processing throughput) and keeps track of the incoming traffic in order to adapt the batching and the batch partitioning.

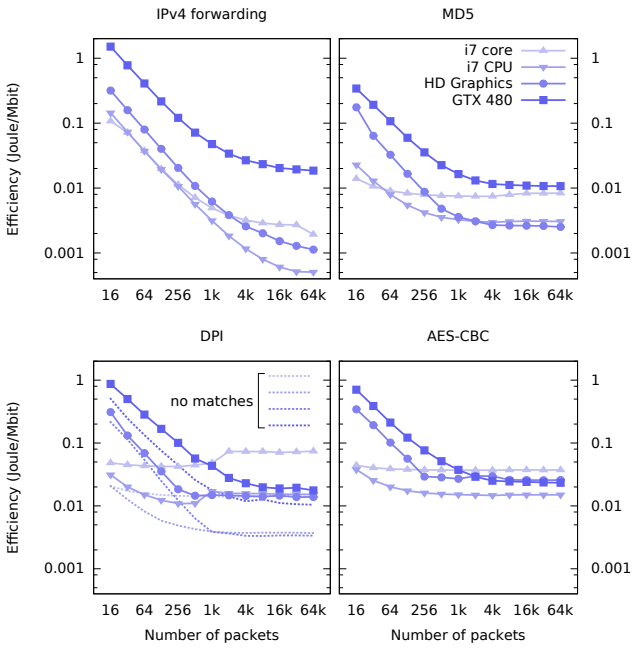


Figure 5: Energy efficiency of different computational devices.

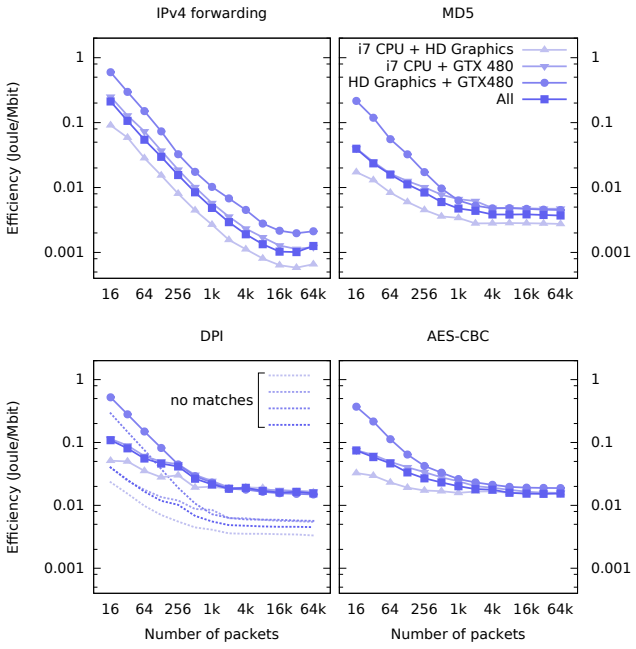


Figure 6: Energy efficiency of different combinations of computational devices.

4.1 Initializing the Scheduler

We first discover the best-performing configuration for each device; we then use these per-device configurations to also benchmark the remaining configurations comprised by combinations of devices. For each combination of our parameter space, we measure the sustained throughput, latency and power, and store them to a dictionary; the dictionary will be used at runtime in order to acquire the most suitable configuration. The time needed to compute the whole table requires 90–360 seconds, using a time quantum of 40 ms or a minimum of two samples, whichever comes last, for each configuration.

We use a different binary tree to store each achieved outcome (i.e. throughput, latency, and power) for each configuration. The motivation behind this is to allow throughput-, latency- and energy-aware applications, to find quickly the most appropriate configuration accordingly. At runtime, the corresponding metric (i.e. throughput, latency, and power) is used to acquire the most suitable configuration. The reason we use a binary tree is to allow fast insertions/updates and (more importantly) support both exact and nearest neighbor searches.

Each node in the binary tree holds *all* the configurations that correspond to the requested result. In order to acquire quickly the most efficient configuration, the configurations, within each node, are further ordered using two additional index structures. Specifically we use one index structure for each of the two remaining requirements. We also reverse the value of power and latency, before normalizing them, as they represent less-is-better metrics. The motivation of using three indices is to allow an application to select *(i)* either the configuration that is best for a single requirement, or *(ii)* the configuration that achieves the best outcome for both requirements. As indices we use priority queues as they can return the element with the maximum weight in $O(1)$.

However, requirements (i.e. throughput, latency, and power) are measured with float precision. As such, exact matches will be very rare, at runtime. To overcome this, we can either round to the smallest integral value (e.g. to the nearest multiple of 100 Mbit/s), or implement support for nearest neighbor search queries. By rounding to the smallest integral value we do not guarantee that a given value should be present in the binary tree; we have to explicitly fulfill the values for all missing neighbors. As the updating of the values of all missing neighbors can be quite costly —especially in sparsely populated cases— we implemented the latter solution. Therefore, if we do not have an exact match, we select the immediately nearest (either smaller or greater) match. Since we utilize a binary tree, the selection of the immediately nearest value can be obtained at the same cost. Moreover, in order to prevent from overloading the binary tree, before inserting a new node in the binary tree, we check if it differs with its parent by a threshold δ. If not, we merge them in order to save space. The threshold δ is also used as a parameter of our adaptation algorithm that is described in the next section.

4.2 Online Adaptation Algorithm

The goal of the online adaptation algorithm is to determine quickly and accurately which device (or devices) is more suitable to sustain the current input traffic rate, and to be able to adapt to changes in the traffic characteristics, with as little overhead as possible. Moreover, it allows an application to get the most suitable configuration based on its own needs (i.e. throughput-, latency-, or energy-critical). For example, it would be better for a latency critical application to submit the incoming traffic to more than one devices, while in an energy-critical setup it would be better to use only the most energy efficient device.

Our scheduling algorithm is laid out as follows. We create a queue for each device, and place packet batches in those queues, according to the following iterative algorithm:

1. Measure the current traffic rate. Get the best configuration from the lookup table, using as search key the desired requirement (i.e. latency-, throughput-, or energy-aware). Change to this configuration only if it was measured better than the current one by a factor of λ. Initialize variables α and β.

2. Start creating batches of the specified size. If more than one devices are required, create batches for each device accordingly. The batches are inserted into the queue of the corresponding device(s).

3. Measure the performance achieved by each of the devices for the submitted batch(es). If the sustained performance is similar to the one requested from the lookup table (up to a threshold δ), return to Step 1; otherwise, update the lookup table accordingly, and:

   - If the performance achieved by each device is lower, increase the batch size by a factor of α; set $\beta = \alpha/2$, and go to Step 3.
   - If the performance achieved by each device is higher, decrease the batch size by a factor of β; set $\alpha = \beta/2$, and go to Step 3.

The scheduler gets continually cross-trained and improves as more network traffic is processed across different devices. Moreover, the scheduler can easily adapt to traffic-, system-, or application-changes. Traffic changes (such as traffic bursts) can easily tolerated by our scheduler by quickly switching to the appropriate configuration (Step 1), without requiring to update the scheduler. In contrast, system- and application-changes should update the scheduler: The loop, that starts at the Step 3 of the adaptation algorithm above, finds the best configuration of the given device for the current conditions. After that, the scheduler returns to Step 1, as more appropriate devices might exist to handle the current conditions. The purpose of the λ factor is to avoid alternating among competing configurations and just maintain a "good enough" state.

Therefore, our scheduler can tackle system changes, such as throttling and contention, that may occur more frequently in the i7 Ivy Bridge processor, where multiple computational devices are integrated into a single package and sharing a single memory system and power budget. Application changes, such as in the case of the DPI which has large performance fluctuations according to the current traffic characteristics are also confronted. To prevent temporal packet loss, in the inter-time that our scheduler needs to adapt to the new conditions, we maintain queues of sufficient size for each device. In Section 5 we show that a few hundred MBs are sufficient to guarantee that no packet loss will occur, during any traffic-, system-, or application-changes.

For the experiments presented in this paper, we set the difference threshold, δ, between the expected and the measured performance to 10%, and the growth and decrease rate, α, β, to 2x; we found that these values provide the best average performance across the set of applications we studied.

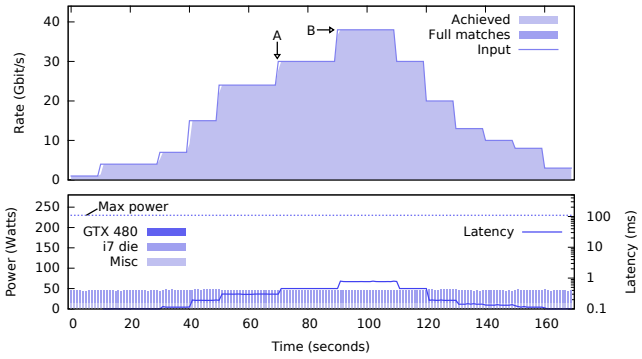
4.2.1 Analysis

The complexity of the algorithm, when searching for the configuration of a specific requirement, is $O(logN)$, where N is the total number of configurations. Indeed, the configurations are stored in a binary tree, hence the searching cost is $O(logN)$. As long as the configurations are found, the cost to acquire the most efficient is $O(1)$, because they are stored in a priority queue. Hence, the overall cost to acquire the most efficient configuration for a given requirement, is $O(logN)$. However, our adaptive algorithm requires that the given configuration should be updated, in case the sustained performance differs by a threshold δ. The update cost is equal to the cost required to find the node in the binary tree ($O(logN)$) and the cost to insert it into the priority queue ($O(logN)$), totalling a $2 \times O(logN)$. After the update, the algorithm converges to the batch size that achieves the requested performance, if any (Step 3). This can take up to $log_\alpha M$ steps (or $log_\beta M$ equivalently), where M is the maximum batch size.

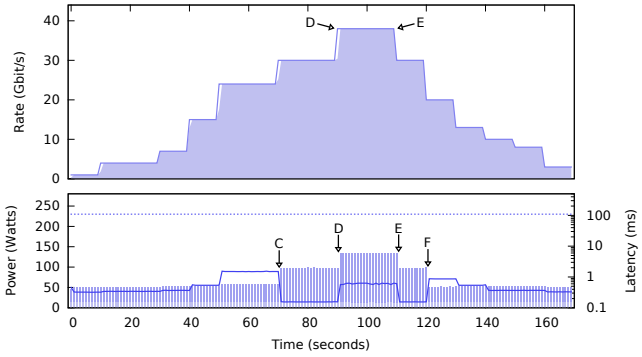
5. EVALUATION

We now evaluate the performance of our scheduling algorithm, using the packet processing applications described in Section 3.2. We use an energy-critical policy, i.e. handle all input traffic at the maximum energy efficiency. In Figure 7(a)–(d) we present the applied and achieved throughput, the power consumption and the device selection made by the scheduler, for the four applications under study. For comparison, we illustrate with a dashed line the power consumption when all three devices are used simultaneously. Additionally, we provide the experienced latency with a solid line. Latency variability is a result of the dynamic scheduler decisions for the batching and computational device selection. The input traffic has the same profile for all applications and is comprised of 25% 60-byte TCP packets and 75% 1514-byte TCP packets.

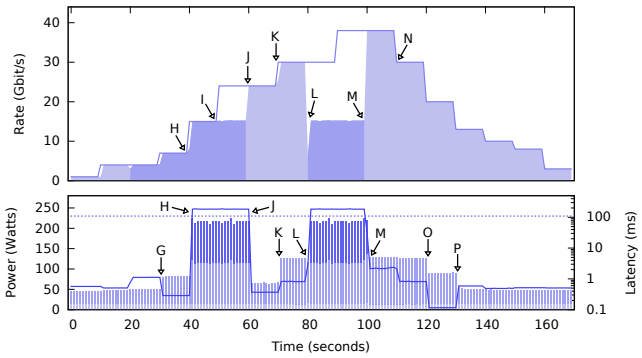
Overall, our scheduler adapts to the highly diverse computational demand among the selected applications, producing dynamic decisions that maintain the maximum energy efficiency during all times. Additionally, it sustains high throughput and avoids excessive latency when possible. Furthermore, our scheduler is able to respond to application specific performance characteristics. For example during DPI (Figure 7(c)), our algorithm detects the requirement for a different configuration at times H and L. H and L introduce packets with a high match rate (in contrast to the low previous match rate), where the target cannot be satisfied without the use of the energy-hungry GTX 480.



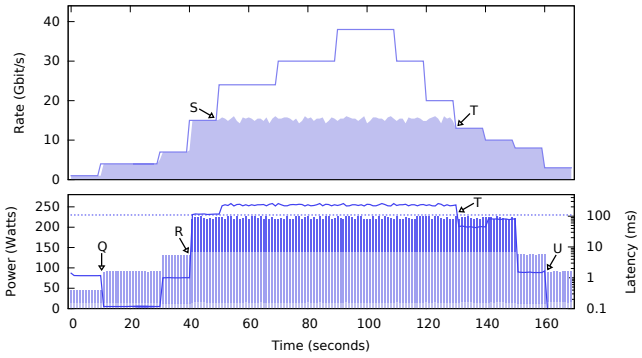
(a) IPv4 Forwarder.



(b) MD5.



(c) DPI.



(d) AES-CBC.

Figure 7: Automatic device configuration selection under different conditions. Optimized for maximum energy efficiency.

5.1 Throughput

We observe that our proposed scheduler is able to switch to the configuration that keeps the selected target, under the required computational capacity which is required to process the incoming traffic for each application. However, there are two cases which our architecture does not sustain the input traffic rate: *(i)* in the DPI application, between I–J as well as L–M, and *(ii)* in the AES application, between S–T. The reason is that there is not a device, or combination of devices, to handle these cases, as we have already seen in Figures 3 and 4. More specifically, the DFA used by the Aho-Corasick algorithm exhibits strong locality of reference when the traffic does not contain any pattern matches; however when the traffic is overwhelmed by, different, pattern matches that locality of reference no longer holds, degrading the overall throughput. The HD Graphics does not handle more than 5 Gbit/s in any case and the GTX 480 performance is restricted by the data transfer bottleneck.

5.2 Energy Efficiency

Our proposed scheduler consistently switches to the most energy efficient configuration at all rates for each application. The advantage of our approach is more noticeable when the load is fairly low (1–4 Gbit/s) as it switches to the energy-efficient integrated GPU. Especially for the IP forwarding, the integrated GPU is able to cope with the input traffic at all rates, providing a constant 50 W consumption, which is *two* times better over the CPU-only and *more than three* times over the discrete GPU only. Packet hashing switches to the CPU when the rate reaches 30 Gbit/s (at C) and then switches to the CPU-HD Graphics pair (at D) in order to handle the 40 Gbit/s input traffic rate. DPI follows a more composite behavior, as it is affected by both the traffic rate and characteristics (i.e. number of matches). Overall, DPI ends up utilizing the two GPU devices when processing full-matches traffic at rates of 15 Gbit/s or higher (H–J and L–M). Nevertheless, when the matches drop to zero, the CPU is able to cope the input traffic (between J–K and O–P); at rates of 30 Gbit/s or higher the system employs the i7 CPU together with the HD Graphics (K–L and M–N). For all other input rates the CPU or HD Graphics alone can sustain the traffic. At L, we synthetically raise the number of matches that results to a temporal fall to 18.7 Gbit/s, before our scheduler considers to also utilize GTX 480 too. With increased rate, while keeping the number of matches at full ratio, we observe that there is no increase in the sustained rate because there is no better configuration available. AES, which is the most computationally intensive application in our set, ends up using all three devices when the traffic rate exceeds 15 Gbit/s (R–T), and are able to handle up to 15.5 Gbit/s rate.

Overall, our scheduler reaches the maximum consumption in the following cases only: *(i)* when the traffic rate exceeds 15 Gbit/s for AES, and *(ii)* when the rate exceeds 15 Gbit/s for DPI, and is overwhelmed with matches. In DPI, interestingly enough, the HD Graphics plus GTX 480 pair is the winner. Overall, our architecture yields an overall energy saving between 3.5 times (IPv4 forwarding) and 30% (AES-CBC) compared to the energy spent when using all three devices.

5.3 Latency

Increasing the batch size results in better sustainable rate at the cost of increased latency, especially for the GTX 480. IPv4 forwarding —executed solely on the HD Graphics— provides a latency that increases linearly with the batch size. However, this is not always the case. For example, in the case of MD5 workload, latency drops significantly in two different time ranges: C–D and E–F. The reason behind this is that the scheduler switches from the HD Graphics to the i7 CPU, in order to handle the increasing traffic rate. Given that the CPU is able handle the requested traffic using a much smaller batch size, results to an extensive latency drop. When the input rate grows further (D–E) though, the HD Graphics is utilized again, together with the CPU. This results to a 3.6 times increase of the measured latency. Similar transitions occur in other workloads as well, e.g. at G and O for DPI, and at Q and U for AES-CBC. We note though that in our experiments we focused primarily on providing a minimum power utilization setup. By using a latency-aware policy, we can obtain much better latency, at the cost of increased power consumption.

5.4 Traditional Performance Metrics

In addition to the previous studied metrics, we measure other significant metrics which are present in the software packet processing domain, namely: packet loss, and reordering. Our algorithm may introduce packet loss by switching to a device too slowly in the face of varying traffic demands. Reordering may be introduced when packets belonging to the same flow are redirected to a different device. Regarding packet loss, our experiments show that our algorithm can react quickly enough to avoid packet drops. We observe that in all cases our proposed scheduler can adapt to changes in less than 300 ms —which is the case where we use the GTX 480 with a batch size of 64K. This roughly results to 1.46 GB of received data (in a 40 GbE network, for a MTU of 1500 bytes), hence a buffer of this size is sufficient to guarantee that no packet loss will occur in the inter-time that our scheduler needs to adapt to the new conditions. We notice however that this is the worst case, in which the input rate goes from zero to 40 Gbit/s and at the same time the algorithm pushes the system to a configuration with the worst latency (300 ms). In our experiments, using a 500 MB buffer was enough.

Finally, we measure packet reordering. In our system, reordering can only occur when traffic is diverted to a new queue. However, as we have described in Section 3.3.1 we ensure that packets with the same 5-tuple will never be placed in batches that will execute simultaneously to different devices. This guarantees that, when using more than one devices, packets of the same flow will always processed by the same device. Indeed, in all our experiments we did not observe any packet reorders.

6. RELATED WORK

Recently, GPUs have provided a substantial performance boost to many individual network-related workloads, including intrusion detection [22, 34, 37, 39, 40], cryptography [20, 23], and IP routing [19]. In addition, several programmable network traffic processing frameworks have been proposed—such as Snap [36] and GASPP [38]—that manage to simplify the development of GPU-accelerated network traffic processing applications. The main difference with these works is that we focus on building a software network packet processing framework that combines different, heterogeneous, processing devices and quantify the problems that arise with their concurrent utilization. By effectively mapping computations to heterogeneous devices, in an automated way, we provide more efficient execution in terms of throughput, latency and power consumption.

A number of recently proposed load-balancing systems support applications with multiple concurrent kernels [15, 35]. Other approaches rely heavily on manual intervention by the programmer [26]. Approaches to load-balance a single computationanl kernel include [13, 24, 27]. The simplest approach target homogeneous GPUs and, thus, require no training as they use a fixed work partition [24]. Wang and Ren propose a distribution method on a CPU-GPU heterogeneous system that tries a large number of different work distributions in order to find the most efficient [41]. Other approaches require a series of small execution trials to determine the relative performance [13, 27]. The disadvantage of these approaches is that they have been designed for applications that take as input constant streaming data and as a consequence, they adapt very slowly when the input data stream varies. That makes them extremely difficult to be applied to network processing applications in which the heterogeneity of *(i)* the hardware, *(ii)* the applications, and *(iii)* the traffic vastly affect the overall efficiency in terms of performance, latency and power consumption. To that end, our proposed scheduling algorithm has been designed to explicitly account for this.

Furthermore, there is ongoing work on providing performance predictability [16] and fair queueing [18] when running a diverse set of applications that contend for shared hardware resources. There is also work on packet routing [29] that draws power proportional to the traffic load. The main difference with our work, is that they focus solely on homogeneous processing cores; instead we present a system that utilizes efficiently a diverse set of devices.

7. LIMITATIONS

Our scheduler requires live power consumption feedback for each of the available computational devices in the system. Even though such schemes have now become common in commodity, off-the-self, processors (e.g. the *Running Average Power Limit interface* present on latest Intel processor series), they are still in a preliminary stage in current graphics hardware architectures (although we expect this to change in the near future). To overcome the lack of such power estimation in current GPU models, we propose the use of a power model as a substance of real instrumentation [21].

Another notable limitation of our proposed architecture is the lack of optimization capabilities for concurrent running applications. The optimal parallel scheduling of an arbitrary application mixture is a highly challenging problem, mainly due to the unknown interference effects. These effects include but are not limited to: *(i)* contention for hardware recourses (e.g. shared caches, I/O interconnects, memory controller, etc.), *(ii)* software resources and *(iii)* false sharing of cache blocks. To make matters worse, the scheduler complexity grows exponentially with the introduction of multiple applications, as the parameter space should be explored for all possible application combinations. As such, in this work we solely focus on optimizing the performance

of a single application that executes on a set of computing devices. As part of our future work we plan to experiment with application multiplexing and investigate the feasibility of a more generic energy-aware scheduler.

8. CONCLUSIONS

In this work we address the problem of improving the efficiency of network packet processing applications on commodity, off-the-self, heterogeneous architectures. Heterogeneous systems can provide substantial performance improvements, but only with appropriately chosen partitioning. Using a static approach can lead to suboptimal performance when the state of traffic, system or application changes. To avoid this, we propose an online adaptive scheduling algorithm, tailored for network packet processing applications, that can *(i)* respond effectively to relative performance changes, and *(ii)* significantly improve the energy efficiency of packet processing applications. Our system is able to efficiently utilize the computational capacity of its resources on demand, resulting in energy savings ranging from 30% on heavy workload, up to 3.5 times for lighter loads.

Acknowledgments

This work was supported by the General Secretariat for Research and Technology in Greece with a Research Excellence grant. Lazaros Koromilas and Giorgos Vasiliadis are also with the University of Crete.

9. REFERENCES

[1] 1018_2 - PhidgetInterfaceKit 8/8/8. http://www.phidgets.com/.
[2] 1122_0 - 30 Amp Current Sensor AC/DC. http://www.phidgets.com/.
[3] OpenCL. http://www.khronos.org/opencl/.
[4] OpenSSL Project. http://www.openssl.org/.
[5] The Snort IDS/IPS. http://www.snort.org/.
[6] Intel 82599 10 GbE Controller Datasheet, Revision 2.0, 2009.
[7] Intel HD Graphics DirectX Developer's Guide, 2010.
[8] Intel SDK for OpenCL Applications 2013: Optimization Guide, 2013.
[9] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese. EndRE: an end-system redundancy elimination service for enterprises. In NSDI, 2010.
[10] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. Communications of the ACM, 18(6):333–340, 1975.
[11] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker. Packet caches on routers: the implications of universal redundant traffic elimination. In SIGCOMM, 2008.
[12] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding Data Center Traffic Characteristics. SIGCOMM CCR, 40(1), 2010.
[13] M. Boyer, K. Skadron, S. Che, and N. Jayasena. Load Balancing in a Changing World: Dealing with Heterogeneity and Performance Variability. In ACM Computing Frontiers, 2013.
[14] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In USENIX Security, 2003.
[15] G. F. Diamos and S. Yalamanchili. Harmony: An Execution Model and Runtime for Heterogeneous Many Core Systems. In HPDC, 2008.
[16] M. Dobrescu, K. Argyraki, and S. Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. In NSDI, 2012.
[17] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In SOSP, 2009.
[18] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-Resource Fair Queueing for Packet Processing. In SIGCOMM, 2012.
[19] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In SIGCOMM, 2010.
[20] O. Harrison and J. Waldron. Practical Symmetric Key Cryptography on Modern Graphics Hardware. In USENIX Security, 2008.
[21] S. Hong and H. Kim. An integrated gpu power and performance model. In SIGARCH, 2010.
[22] M. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. Kargus: a Highly-scalable Software-based Intrusion Detection System. In CCS, 2012.
[23] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In NSDI, 2011.
[24] J. Kim, H. Kim, J. H. Lee, and J. Lee. Achieving a single compute device image in OpenCL for multiple GPUs. In PPoPP, 2011.
[25] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of files. In USENIX ATC, 2004.
[26] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: A Programming Model for Heterogeneous Multi-core Systems. In ASPLOS, 2008.
[27] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In MICRO, 2009.
[28] G. Maier, A. Feldmann, V. Paxson, and M. Allman. On dominant characteristics of residential broadband internet traffic. In IMC, 2009.
[29] L. Niccolini, G. Iannaccone, S. Ratnasamy, J. Chandrashekar, and L. Rizzo. Building a Power-Proportional Software Router. In USENIX ATC, 2012.
[30] NVIDIA. CUDA C Programming Guide, Version 5.0, 2012.
[31] R. Rivest. The MD5 message-digest algorithm. 1992.
[32] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In USENIX ATC, 2012.
[33] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu. Performance Traps in OpenCL for CPUs. In PDP, 2013.
[34] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating GPUs for Network Packet Signature Matching. In ISPASS, 2009.
[35] E. Sun, D. Schaa, R. Bagley, N. Rubin, and D. Kaeli. Enabling Task-Level Scheduling on Heterogeneous Platforms. In GPGPU, 2012.
[36] W. Sun and R. Ricci. Fast and Flexible: Parallel Packet Processing with GPUs and Click. In ANCS, 2013.
[37] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In RAID, 2008.
[38] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis. GASPP: A GPU-Accelerated Stateful Packet Processing Framework. In USENIX ATC, 2014.
[39] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In RAID, 2009.
[40] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. MIDeA: a multi-parallel intrusion detection architecture. In CCS, 2011.
[41] G. Wang and X. Ren. Power-efficient work distribution method for cpu-gpu heterogeneous system. In ISPA, 2010.