Hermes: An Integrated CPU/GPU Microarchitecture for IP Routing

Yuhao Zhu
Department of Electrical and Computer Engineering
The University of Texas at Austin
yuhao.zhu@mail.utexas.edu

Yangdong Deng
Institute of Microelectronics
Tsinghua University
dengyd@tsinghua.edu.cn

Yubei Chen
Institute of Microelectronics
Tsinghua University
chen-yb08@mails.tsinghua.edu.cn

ABSTRACT

With the constantly increasing Internet traffic and fast changing network protocols, future routers have to simultaneously satisfy the requirements for throughput, QoS, flexibility, and scalability. In this work, we propose a novel integrated CPU/GPU microarchitecture, Hermes, for QoS-aware high speed routing. We also develop a new thread scheduling mechanism, which significantly improves all QoS metrics.

Categories and Subject Descriptors

C.2.6 [**Computer-Communication Networks**]: Internetworking – *Routers*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors) – *Single-instruction-stream, multiple-data-stream processors (MIMD)*; C.1.3 [**Processor Architectures**] Other Architecture Styles – *Heterogeneous (hybrid) systems*

General Terms. Performance

Keywords

Software Router, QoS, CPU/GPU Integration

1. Introduction

As the backbone of Internet, IP routers provide the physical and logic connections among multiple computer networks. When a packet arrives, a router will determine which out-bounding network the packet should be forwarded to according to the current routing table and the packet's destination IP address. On modern routers, typical packet processing involves a series of operations on packet headers/body [1] by a CPU or a network processor. A router has to deliver a high forwarding throughput under stringent quality-of-service requirements. In this work, we focus on improving packet processing performance by developing a novel integrated CPU/GPU microarchitecture as well as a corresponding QoS-aware scheduling mechanism.

Now IP routers have to face a unique set of challenges. First of all, the Internet traffic is still exponentially increasing, especially with the introduction of on-line video and P2P technologies (e.g. [2]). In fact, current core routers are required to deliver a throughput of 40Gbps - 90Tbps [3]. Another trend is that new network services and protocols are fast appearing, although the Internet is still based on IPv4 and Ethernet technologies that both were developed in 1960s [4]. As a result, the original protocols have to be extended and updated to adapt to today's network applications. Good programmability is crucial to meet such requirements.

Current router solutions can be classified into application-specific integrated circuit (ASIC) based, network processor based, and software based [5]. However, none of these solutions could simultaneously meet the requirements for both high throughput and programmability.

Recently graphic processing units (GPUs) are appearing as a new platform that could offer strong computing power [6]. GPUs also have a stable mass market and thus strong supports for software development (e.g., [7] and [8]). Accordingly, it is appealing to leverage the mature GPU microarchitecture for network routing processing. Two recent works already proved the performance potential of GPU to accelerate packet processing [9] and [10]. Nevertheless, it is also found that current GPU architectures are still under serious limitations for routing processing. First, GPU computing requires the packets to be copied from CPU's main memory to GPU's video memory. The extra memory copy introduces a performance overhead. Second, the batch based GPU processing could not guarantee processing QoS for an individual packet, although such QoS requirements are critical for routers. In this work, we develop novel solutions to augment an existing GPU microarchitecture for high speed packet processing with QoS assurance. Our basic design philosophy is to make the best out of mature hardware and software solutions with minimal modifications. The contributions of this paper are as follows.

- We proposed Hermes, an integrated CPU/GPU, shared memory microarchitecture that is enhanced with an adaptive warp issuing mechanism for IP packet processing. To the best of our knowledge, this is the first work on developing a GPU-based packet processing platform that simultaneously optimizes all metrics of QoS.
- A complete set of router applications were implemented on Hermes. We also conducted extensive QoS evaluations on Hermes microarchitecture. When compared with a GPU-accelerated software router [9], Hermes delivers a 5X enhancement in throughput, an 81.2% reduction in average packet delay, as well as a 72.9% reduction in delay variance.

The rest of the paper is organized as follows. Section 2 reviews the background of this work. Section 3 details the hardware and software designs of the Hermes microarchitecture. A thorough performance evaluation is presented in Section 4. Section 5 concludes the whole paper and outlines important future research directions.

DAC'11, June 5-10, 2011, San Diego, California, USA

2. Background and Motivation

The responsibility of a router is to deliver packets from ingress ports to egress ports in a timely manner. The fundamental tasks of IP layer packet processing involves checking IP header, packet classification, routing table lookup, decrementing TTL value, and packet fragmentation. The critical path of a complete IP router tasks chain is presented as follows: Checking IP Header (CheckIPHeader) → Packet Classification (Classifier) → Routing Table Lookup (RTL) → Decrementing TTL (DecTTL) → IP Fragmentation (Fragmentation) [4]. In addition, to meet the ever-demanding requirements for intrusion detection, deep packet inspection (DPI) [11] is increasingly becoming a regular task deployed before routing processing or even an integral part of modern IP routers. We use all the above tasks as benchmark applications in our experimental evaluations.

2.1 Current IP Router Solutions

Modern IP router solutions can be classified into three main categories, hardware routers, software routers and programmable network processors (NPUs) [5].

Hardware routers depend on customized hardware, i.e., ASICs, to deliver the highest performance with the least power/area overhead. Nonetheless, hardware routers suffer from the long design turnaround time, high non-recurring engineering (NRE) cost, and poor scalability and programmability. Such hurdles have made ASIC based solutions gradually out of the mainstream routers.

In contrast, software routers implement all packet processing applications as programs running on commodity computers (e.g., [12]). They are extremely flexible because any change in network configurations and protocols can be realized through reprogramming. Furthermore, general purpose processors are targeting a much more massive market and thus backed up with more mature operating systems and development tools. However, it is extremely challenging for pure software routers to deliver sufficient processing power required by high performance networks. Typically, software routers could only deliver a throughput of 1-3Gbps, which is considerably lower than the required throughput of 40Gbps - 90Tbps for core networking equipment [3]. Therefore, such routers can only be used in relatively small networks.

In the middle of the solution spectrum is the network processing unit (NPU) based IP routers. NPUs are dedicated packet processing engines by integrating a given number of identical processing elements designed for packet manipulation [13]. A clear downside of NPUs is that so far no effective programming models have been constructed due to the limited size of market and customer base [14]. The small volume of NPUs also leads to prohibitive per-chip cost. In addition, network processors often need customized logic modules for critical path processing [3]. Such modules could not be reprogrammed for new network protocols. The above hurdles already forced some top NPU vendors to close their product line of network processor [15] and resort to multi-core based router solutions (e.g., [16]).

2.2 GPU Architecture and Programming Model

Originally designed for graphics acceleration, GPUs are recently emerging as a high performance general-purpose computing platform [6]. A typical GPU is organized as an array of multiprocessors or shader cores. Each shader core will deploy multiple streaming processors (SPs) as well as a small amount of software controlled shared memory. A GPU program deploys a large number of threads organized into blocks with each block assigned to a unique shader core. A shader core would then decompose a block of threads into 32-thread warps. A warp is the basic unit of job scheduling on GPUs. Each warp will always follow the same instruction schedule with each thread handling a different data set. In other words, a warp of threads would execute instructions in a single-instruction, multiple data (SIMD) fashion. GPUs are supported with a video memory, or global memory in NVIDIA's CUDA terminology. The global memory offers a high memory bandwidth, but also incurs a long latency.

2.3 Pros and Cons of GPU Based Packet Processing

The Internet services are realized through a hierarchical organization of packet processing. Generally the processing of a packet is independent with others. Such a fundamental observation suggests that GPU is potentially a good packet processing engine because various SP cores could handle multiple packets in parallel. Two recent works reported in [9] and [10] already proved the potential of GPUs for packet processing. It is shown that a GPU based software router solution could outperform a CPU baseline router by a factor of up to 30X.

However, two main problems still need to be resolved before a GPU accelerated software routers can be practical. First of all, the communication mechanism between CPU and GPU seriously degrades system throughput. In fact, the packets arriving at the router are first copied to CPU main memory and then to GPU global memory through a PCI Express (PCIe) bus with a peak bandwidth of 8GB/s [17]. The extra memory copy introduces performance and power overhead. The situation is exemplified by a signature matching application reported in [9]: the pure processing throughput of GPU can be over 30 times higher than that of CPU, but the speed-up degrades to 5X when considering the data transfer overhead.

Secondly, the GPU's batch processing model introduces a "throughput vs. delay" dilemma. The design philosophy of GPU is to employ the parallel architecture to maximize the overall throughput. Therefore, to best leverage the computing power of GPU, it is beneficial to accumulate data for sufficient parallelism. For packet processing, it means that the CPU needs to buffer enough packets before they can be transferred to GPU for parallel processing. Such a mechanism could worsen the latency for certain packets. Suppose each time *batch_transfer_granularity* bytes of data between CPU and GPU at a line card rate of *line_card_rate* of bytes per second. Then under the extreme condition, the earliest arrived packet in a buffer has to wait for a time of *batch_transfer_granularity / line-card_rate* before it can be served. In addition, the GPU programming model typically organizes threads into blocks for batch processing. This organization further exacerbates the worst-case delay because a block finishes execution only when all internal threads have completed.

3. Hermes System

In this work, we developed Hermes, a heterogeneous microarchitecture with CPU and GPU integrated on a single chip. The memory copy overhead can thus be removed by sharing a common memory system between CPU and GPU. On top of such a microarchitecture, we proposed an adaptive warp issuing mechanism to reduce worst-case packet latency. In this section, we first discuss the underlying microarchitecture, and then detail the adaptive warp issuing mechanism as well as its corresponding software extensions.

3.1 CPU/GPU Integration with Shared memory

With the rapidly growing integration capacity made available by the advancement of semiconductor process, it is now feasible to

integrate closely-coupled CPU and GPU cores on a single chip (e.g., [18]). The CPU and GPU access a common memory structure including both caches and DRAMs. Since Hermes is designed for high-performance network processing, we choose to use GDDR5 [19] DRAM chips for better bandwidth. A diagram of Hermes is shown in Figure 1.

The overall execution flow remains the same as a classical heterogeneous CPU/GPU system (e.g., [9][10]). The CPU is responsible for creating and initializing data structures according to packet processing applications. Upon the arrival of network packets, the packet data are stored into the shared memory and then fetched by shader cores for processing. Finally, the contents of the processed packets are updated in the shared memory, where they can be either further processed by CPU or directly forwarded to the destinations.

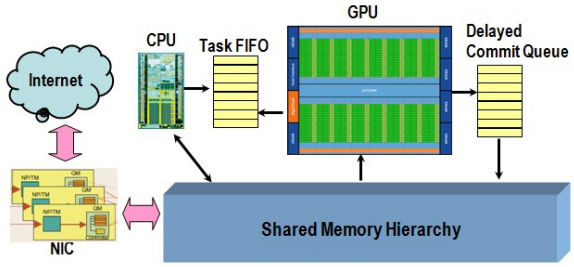

Figure 1. Hermes microarchitecture

The shared memory system serves as a large packet buffer to avoid the classical buffer sizing problem (e.g., [20]). In a typical network environment, the rule of "Bandwidth-Delay Product" (BDP) [21] mandates a 1.25GB buffer size [22], which is impractical in traditional router designs. On the other hand, routers using smaller buffers suffer from high packet loss rate [23]. However, the shared memory space in Hermes is naturally large enough to hold a sufficient number of incoming packets (even in case of burst) to guarantee packet availability.

The programming model of Hermes is compatible with NVIDIA CUDA. Besides taking advantaging of existing development tools, such a programming model also avoids the memory coherency problem that is typical in shared memory architectures. Actually, CPU and GPU independently access data in CUDA. In case of packet processing, the CPU and GPU operations on an individual packet would be mutually exclusive. Nevertheless, it must be noted that the out-of-order commitment of processed packets could introduce coherency problems, which will be further discussed in the next section.

3.2 Adaptive Warp Issuing Mechanism

The warp issuing mechanism of Hermes is responsible for assigning parallel tasks onto shader cores for further intra-core scheduling. On current GPUs, all the thread warps are kept in a warp pool before being issued. In order to maximize the overall throughput, warps are issued to shader cores by following a best-effort strategy, which means the number of warps that can be issued in one round is only constrained by the number of available warps as well as hardware resources such as per core register and shared memory size. For packet processing applications, however, it can be unaffordable to wait for an enough number of warps under certain circumstances. Therefore, we proposed an adaptive warp issuing mechanism that adapts to the arrival pattern of network packets and maintains a good balance between overall throughput and worst-case per-packet delay.

3.2.1 Mechanism

The packets are received by network interface cards (NICs) and then copied to the shared memory via DMA transfers. The CPU is thus able to keep track of the number of arrived packets. Accordingly, the CPU is responsible for notifying the GPU to fetch packets for processing. As illustrated in Figure 1, a simple task FIFO would suffice to support such an interaction.

When CPU decides it is appropriate to report the availability of packets, it creates a new FIFO entry with its value as the number of packets ready for further processing, assuming the task FIFO is not full. Meanwhile, the GPU is constantly monitoring the FIFO and making decisions on fetching a proper number of packets. Of course, the minimum granularity, i.e., number of packets, of one round of fetching by GPU should be at least equal to the number of threads in one warp. Otherwise, the GPU hardware is not fully utilized.

One essential question is that how frequently the CPU should update the task FIFO. It directly relates to the transferring pattern from NIC to shared memory. Again here a tradeoff has to be made. On the one hand, transferring a packet from NIC to the shared memory involves a book-keeping overhead such as reading and updating the related buffer descriptors. The corresponding extra bus transactions may be unaffordable [24]. In addition, too frequently updating of the task FIFO also complicates GPU fetching due to the restriction of finest fetching granularity mentioned before. On the other hand, too large an interval between two consecutive updates increases average packet delay. Moreover, setting a lower bound on transfer granularity, in the worst case, results in a timeout problem, which would postpone the processing of some packets. If such a timeout really happens, the NIC logic and corresponding device drivers have to be equipped with a timeout recovery mechanism. Considering the contradicting concerns, we set the minimum data transfer granularity to be the size of a warp, i.e., 32 packets. In addition, if there are not enough packets arriving in a given interval, these packets should still be fetched and processed by GPU. A similar approach is taken by [24]. The interval is chosen to be 2 warp arriving times, although it is should be based on the packet arriving rate, i.e., an adaptive estimation. Upon finishing one transaction of data transfer (from NIC to system memory), the CPU notifies GPU through updating the FIFO. Such a configuration generally guarantees that there are enough packets to be processed in one warp.

Newly created warps are put in a pool for issuing. A round-robin issuing strategy is employed to evenly distribute the workload among each shader core. Clearly it is necessary to keep track of the availability of hardware resource that eventually restricts the maximum number of concurrently active warps. Upon reaching the limit, the warp issuing should pause until new processing slots are available.

3.2.2 Guaranteed In-Order Warp Commit

With the fine-grained multithreading execution model, thread warps running on one shader core may finish in an arbitrary order, not to mention warps running on different shader cores. As a result, sequentially arrived packets could get processed with any order. Typically, TCP does not enforce the order of packets processing and committing, since its header includes extra areas to enable retransmission and reassembly, but UDP does require in-order processing [25]. Therefore, it could be mandatory to maintain the packet commitment order the same as the arriving order to guarantee compatibility among different protocols. In NPUs, a complicated task scheduler is responsible for this pur-

pose [26], but our solution only relies on a simple Delay Commit Queue (DCQ).

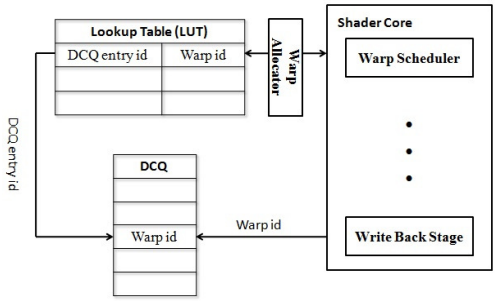

Figure 2. Implementation of Delay Commit Queue (DCQ) with a Lookup Table (LUT)

The key idea is to allow out-of-order warps execution but enforce in-order commitment. This resembles the Reorder Buffer (ROB) in a processor with hardware-enabled speculation mechanism, although our approach requires much less hardware cost.

As illustrated in Figure 2, the DCQ holds the IDs of those warps that have finished but not committed yet. Every time a warp is about to be issued onto one shader core, and the DCQ is not full, a new entry is allocated. This mapping between warp ID and its DCQ entry ID is recorded in a lookup table (LUT). Upon finishing, the corresponding DCQ entry is updated by indexing the LUT with the finished warp's ID. Only could a warp be committed when all warps arrived earlier have been finished. Once a warp commits, its DCQ entry is reclaimed.

Interestingly, the DCQ also prevents hazards on packet data. The CPU should be aware of the warp status to perform further operation. Such information could be reflected by the header pointer of DCQ. Without DCQ, the write operation by GPU and the read operation of CPU may lead to a data hazard.

3.2.3 Hardware Implementation and Cost Estimation

Since Hermes is implemented by augmenting an existing GPU microarchitecture, it is necessary to evaluate the extra hardware cost required. As mentioned earlier, we will need three new components, task FIFO, Delay Commit Queue, and DCQ-Warp LUT storing the mapping of warp index to DCQ entry.

Both task FIFO and Delay Commit Queue need to be assigned with a finite size, but there is no theoretical upper bound that could always avoid overflow. Based on extensive experiments, we find that a size of 1K entries for both task FIFO and DCQ suffice for typical packet traces. Since task FIFO stores the number of newly arrived tasks, its entries can be set with a size of one integer, i.e. 32 bits. The DCQ records the warp index and thus the size of its entries can be set as the total number of maximally allowed concurrent warps (MCWs) over all shader cores. In our work, the number of MCWs in a shader core is no larger than 32. Assuming 8 shader cores installed on GPU, the DCQ's entry size can be set as 8 bits. For the DCQ-Warp LUT in a shader core, the number of its entries equals to the number of MCWs. Therefore, one LUT should have 32 entries, with each entry having a warp index portion of 5 bits and a DCQ index portion of 10 bits (to identify a unique entry in DCQ). To ease the alignment issues, we use 16 bits for each entry. Altogether, for a GPU with 8 shader cores, we will need 5.5KB of extra storage that should be implemented in SRAM.

We use CACTI 5.1 [27] to estimate the area cost. Assuming a 45nm process, CACTI's SRAM model reports that task FIFO and DCQ cost $0.053mm^2$ and $0.013mm^2$ respectively, while 8 DCQ-Warp LUTs take $0.006mm^2$ in total. Compared to the total area of one GPU chip, the hardware overhead is next to negligible.

3.3 API Modifications

To support the shared memory system and adaptive warp issuing, Hermes requires a few minor modifications on the host side API. A new built-in variable is also required for GPU kernels. Currently implemented as a library, all modifications are based on the CUDA programming model [6], and thus can be integrated into CUDA native language and runtime system.

With the CPU and GPU sharing the same memory storage, the explicit memory copy is not necessary. Therefore, we do not need the memory copy APIs any more. Instead, we add two new memory management APIs, *RMalloc(void **, size_t)* and *RFree(void *)*, for allocating and freeing memory storages.

In addition, we also abandon the concept of Cooperative Thread Array (CTA), or thread blocks, and directly organized threads in warps. In fact, it is equivalent to regard CTA size in Hermes as always equal to warp size, i.e., 32. The reason for removing such a thread hierarchy is two-fold. First, we want to reduce the granularity of GPU batch processing to achieve a better average packet delay. Second, in packet processing applications, it is naturally unnecessary for different packets (threads) to share data. Therefore, under such a situation it does not need to organize threads into a larger CTA. An implication of the above decision is that we are now unable to compute a unique index for every thread (packet) as in common CUDA practices, i.e., *unique_id = blockIdx * blockDim + threadIdx*. Instead, we define a new built-in variable, *packetIdx*, which is the only piece of necessary information needed to program kernel codes.

4. Experimental Evaluation

In this work, we use GPGPU-Sim [28], a cycle-accurate GPU microarchitecture simulator that supports CUDA programs, to evaluate our modifications. The GPU microarchitecture configurations used in this work are presented in Table 1.

Table 1. Architectural Parameters

Hardware Structure	Configuration
# Shader cores	8
SIMD width	32
Warp size	32
Shader core frequency	1000MHz
# Registers per shader core	16768
Shared memory size per shader core	16KByte
Maximally allowed concurrent warps per core	User defined

It is worth noting that, in the current implementation of GPGPU-Sim, the host side CUDA codes run on a normal CPU, while the kernel codes are parsed and executed on the simulator. In other words, GPGPU-Sim can only evaluate the performance evaluations of GPU computations. To avoid the complexity and performance overhead of integrating a CPU simulator with GPGPU-Sim, we evaluated the performance advantage of Hermes merely in terms of the overhead of PCIe transfers, which clearly dominate in a GPU accelerated software router like the one proposed in [9].

To evaluate Hermes, we implemented a complete CUDA-enabled software router, which covers all the tasks as declared in Section 2. For DPI, a bloom filter based algorithm [29] that is amenable for GPU implementation is employed. We take the string rule sets from Snort [30], and replay network traffic by Tcpreplay [31]. Packet traces for routing table lookup are retrieved from

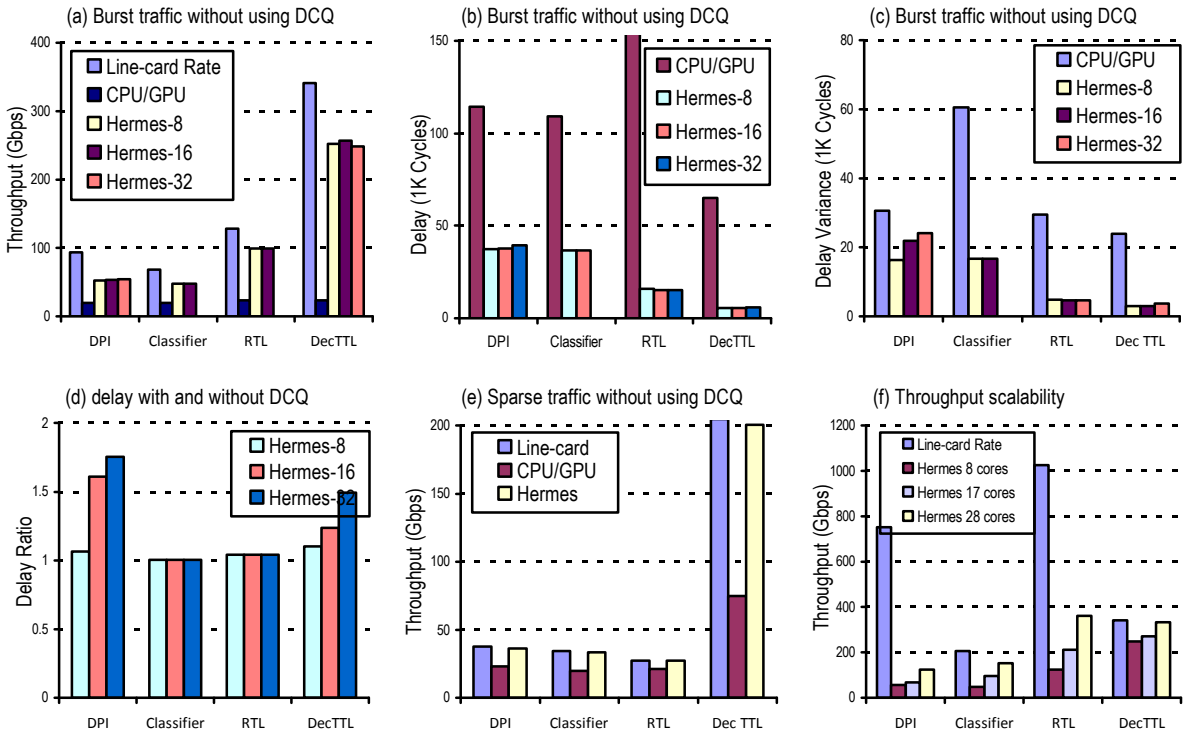

Figure 3. QoS metrics (a) throughput under burst traffic, (b) delay under burst traffic, (c) delay variance under burst traffic, (d) delay ratio, (e) throughput under sparse traffic, and (f) throughput scaling with # shader cores.

RIS [32]. Packet classification implements the basic linear search algorithm and uses ClassBench [33] as the benchmark. The other three applications (CheckIPHeader, DecTTL, and Fragmentation) are adapted from RouterBench [34], and tested under WIDE traffic traces [35].

Since Hermes is targeting the IP routing applications, QoS is of key importance when evaluating system performance. QoS can be generally measured in terms of the following four major metrics, throughput, delay, delay variance, and availability [36]. Because we always perform loss-free tests, we omit the availability metric and only report results of the other three.

Throughput is defined as the total number of bits that can be processed and transferred during a given time period. Delay for a given packet is the time interval between the moment it enters the router and the time it is processed. It consists of both queuing delay and service delay. When the packet arriving rate (line-card rate) exceeds the processing throughput of the system, the succeeding packets have to wait before shader cores are available. This waiting time is the queuing delay. The service delay is the time for a packet to receive complete processing by a shader core (Note that the time spent in the DCQ is included). The delay disparity of different packets measured by interquartile range is designated as delay variance. In our experiments, we regard one warp as the minimum granularity of packet processing and omit the negligible variance among different packets in one warp.

According to our profiling, DPI, packet classification and routing table lookup together consume nearly 90% of total processing time while the rest three (CheckIPHeader, DecTTL and Fragmentation) are much less demanding for processing power. In addition, the latter three applications have almost identical behaviors. Therefore, we use DecTTL as a representative.

Figure 3 shows the three QoS metrics of the four benchmark applications. The number of maximally allowed concurrent warps (MCW) as well as the line-card rate are tuned to get different QoS outcomes. We also present the influence of delay commit queue and the number of shader cores. It is worth noting that due to the limitation of available registers, #MCW cannot be set to 32 for the packet classification application.

A burst traffic that requires packet to be buffered before serviced is used in Figure 3(a) to 3(d). A sparse traffic is applied in 3(e). Both burst and sparse traffics are used in 4(f). Each application has their own line-card rate provided by traffic traces, as shown by the leftmost column of four column-sets in 3(a), 3(e) and 3(f).

Figure 3(a) compares a traditional CPU/GPU system with different configurations of Hermes. The overall processing time of CPU/GPU system consists of three components, packet-waiting overhead (the waiting time before enough packets are available for processing), PCIe transfer time, and GPU computation time. Hermes removes the PCIe transfer overhead and amortizes the packet-waiting time among computation. Therefore, the average throughput of Hermes can still outperform CPU/GPU by a factor of 5 in the best case, although the adaptive issuing mechanism violates the throughput-oriented design philosophy of GPU.

Hermes can deliver network packets with a much smaller delay than a traditional CPU/GPU system as showed in 3(b). On the CPU/GPU system, packet delay is composed of waiting delay on the CPU side as well as processing delay on the GPU side. For RTL and DecTTL, due to their relatively simple processing on GPU, the waiting overhead at CPU side would contribute to a non-negligible part of total delay. Therefore, delay improvement is more significant for these two applications, since Hermes can overlap CPU side waiting overhead with GPU processing. Com-

paring different configurations of Hermes, Hermes-8 (i.e., #MCW equals 8, and so on) always performs the worst. Classifier and RTL do not present significant difference for other three configurations. On average, the best case of Hermes can reduce packet delay by 81.2%.

As showed in 3(c), Hermes also outperforms CPU/GPU system in delay variance by 72.9% on average. Interestingly, the delay variance displays a similar trend as the delay itself. This indicates that the tendency of packet processing is consistent over all delay values.

Although the using of DCQ will not affect the overall throughput, Figure 3(d) shows its impact on packet delay by normalizing to the corresponding cases without DCQ. The DCQ always results in longer packet delay, especially for DPI and DecTTL. It is because those packets taking divergent branches consume much longer time than those following convergent branches in these 2 applications. The longer processing time mandates later-arrived packets buffered in DCQ, deteriorating average delay.

We also perform a sparse traffic test where the arriving rate of packets is lower than the computing rate of Hermes shader cores. As illustrated in Figure 3(e), now packets can be issued without being queued. Therefore, they can be finished almost at the arriving rate, only penalized by the transfer overhead. Even in this case, the CPU/GPU system [9] is still unable to deliver the packets at their arriving rate.

Finally, 3(f) demonstrates the scalability of the Hermes system. With the increasing of the number of shader cores from 8 to 17 to 28 by changing the mesh configuration in GPGPU-Sim from 4x4 to 5x5 to 6x6, the overall performance scales accordingly. Note that in DecTTL, the scaling factor does not completely follow that of shader cores. It is because the arriving packet rate is too sparse for 28-core Hermes so that shader cores are not fully utilized, as justified by the fact that in DecTTL the throughput of Hermes28 is approximately equal to line-card rate.

5. Conclusion and Future Work

In this work, we proposed an integrated CPU/GPU microarchitecture with a QoS-aware GPU scheduling mechanism for accelerating IP routing processing. A complete set of router applications were implemented on this architecture. Experimental results proved that the new microarchitecture could meet stringent delay requirements, while at the same time maintain a high processing throughput. Through minimal augmentation on the current GPU microarchitecture, this work opens a new path toward building high quality packet processing engines for future software routers.

In the future, we will first explore the possibility of a better communication mechanism between NIC and Hermes since now it turns out to be another performance bottleneck. In addition, we are exploring a hardware and software framework to exert both task and data level QoS control on GPU-like microarchitectures.

6. References

[1] F. Baker, Requirements for IP Version 4 Routers, Internet RFC 1812, June 1995.

[2] E. Schumacher-Rasmussen, Cisco Predicts Video Will Make Up 91% of all Internet Traffic by 2014, http://www.streamingmediaeurope.net/2010/06/02/cisco-predicts-video-will-make-up-91-of-all-internet-traffic-by-2014, 2010.

[3] W. Eatherton, The Push of Network Processing to the Top of Pyramid, Keynote Speech at ANCS, 2005.

[4] L. De Carli, et. al., PLUG: Flexible Lookup Modules for Rapid Deployment of New Protocols in High-speed Routers, In Proc. of SIGCOMM, 2009.

[5] H. J. Chao and B. Liu, High Performance Switches and Routers. Wiley-Interscience. 2007.

[6] D. Blythe. Rise of the Graphics Processor. In Proc. of IEEE, vol. 96, No. 5. 761– 778, 2008.

[7] NVIDIA, CUDA Programming Guide 2.3. 2009.

[8] J. Hensley, AMD CTM overview. In International Conference on Computer Graphics and Interactive Techniques, 2007.

[9] S. Mu, et al. IP Routing Processing with Graphic Processors. In Proc. of DATE, 2010.

[10] S. Han, et al. PacketShader: a GPU-Accelerated Software Router. In Proc. of SIGCOMM, 2010.

[11] G. Varghese. Network Algorithmics. Elsevier/Morgan Kaufmann. 2005.

[12] E. Kohler, et al., The Click Modular Router. ACM Trans. On Computer Systes. Vol. 18, No. 3, 2000.

[13] M. Peyravian, and J. Calvignac. Fundamental Architectural Considerations for Network Processors. In International Journal of Computer and Telecommunications Networking. 41(5), April 2003.

[14] C. Kulkarni, et al. Programming Challenges in Network Processor Deployment, In Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp. 178-187, 2003.

[15] R. Merritt. Intel Shifts Network Chip to Startup. EE Times. http://www.eetimes.com/news/latest/showArticle.jhtml?articleID=202804472. 2007.

[16] Intel Whitepaper. Packet Processing with Intel Multi-Core Processors. 2008.

[17] PCI SIG. (2007). PCI Express Base 2.0 specification. http://www.pcisig.com/specifications/pciexpress/.

[18] AMD. The AMD Fusion™ Family of APUs. http://sites.amd.com/us/fusion/APU/Pages/fusion.aspx.

[19] Wiki, GDDR5, http://en.wikipedia.org/wiki/GDDR5.

[20] D. Wischik, and N. McKeown. Buffer Sizes for Core Routers. ACM SIGCOMM Comp. Communications Review, July 2005.

[21] C. Villamizar and C. Song. High Performance TCP in ANSNet. ACM SIGCOMM Comp. Communications Review, 24(5):45-60, 1994.

[22] A. Vishwanath, et al. Perspectives on Router Buffer Sizing: Recent Results and Open Problems. ACM SIGCOMM Comp. Communications Review, April 2009.

[23] A. Dhamdhere, and C. Dovrolis. Open Issues in Router Buffer Sizing. ACM SIGCOMM Comp. Communications Review, Jan. 2006.

[24] N. Egi, et al. Understanding the Packet Processing Capability of Multi-Core Servers. Intel Technical Report.

[25] J. Postel, User Datagram Protocol, Internet RFC768, August 1980.

[26] T. Wolf, and M.A. Franklin, Locality aware predictive scheduling of network processors, In Proc. of ISPASS 2001.

[27] S. Thoziyoor, et al. CACTI 5.1. Technical Report HPL-2008-20, Hewlett Packard Laboratories Palo Alto, April 2008.

[28] A. Bakhoda, et al. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In Proc. of ISPASS, 2009.

[29] B. Bloom. Space/time trade-offs in hash coding with allowable errors. Communication of the ACM, vol. 13, pp. 422-426, Jul. 1970.

[30] The Snort Project, Snort users manual 2.8.0. http://www.snort.org/docs/snort/manual/2.8.0/snort manual.pdf.

[31] Tcpreplay. http://tcpreplay.synfin.net/trac/.

[32] Routing Information Service (RIS). http://www.ripe.net/projects/ris/rawdata.html.

[33] ClassBench: A Packet Classification Benchmark. http://www.arl.wustl.edu/classbench/index.htm.

[34] Y. Luo, et al, Shared Memory Multiprocessor Architectures for Softwre IP Routers, IEEE Transaction On Parallel and Distributed Systems, Vol.14, No. 12, Dec. 2003.

[35] MAWI Working Group Traffic Archive. http://mawi.wide.ad.jp/mawi/

[36] P. M., IP Quality of Service, Helsinki University of Technology, Laboratory of Telecommunications Technology, 1999.