A New Heterogeneous Packet Processing Architecture and Its Analytical Performance Model

Yuhao Zhu, Yubei Chen, and Yangdong Deng

Abstract— Today's IP routers have to simultaneously meet multiple requirements such as programmability, scalability, power, and price besides the traditional objective of high throughput. Software routers like Click offer the best flexibility but suffer from a lower level of processing throughput. A few recent works prove the potential of Graphic Processing Units (GPUs) for high-speed packet processing. However, current GPU architectures cannot guarantee quality-of-service (QoS) of IP routing due to the batched execution model. In this work, we propose a novel heterogeneous, integrated CPU/GPU microarchitecture, Hermes, which adaptively maintains a balance between packet latency and overall throughput. A complete set of router applications are implemented on this architecture. Experimental results show that Hermes achieves a 5X enhancement in throughput, a 81.2% reduction in average packet latency, and a 72.9% reduction in delay variance, when compared with a GPU accelerated software router. A byproduct of this research is an analytical model that catpures the throughput and processing latency of Hermes-alike heterogensous architecture. In this work, the model can be used to quickly estimate IP routing QoS metrics of different Hermes microarchitectural configurations under various traffic patterns. Simulation results reveal that the analytical model can accurate predict QoS metrics with an average error rate of less than 10%.

Index Terms—Routers, SIMD processors, Heterogeneous (hybrid) systems.

1 Introduction

Recent years have seen a strong momentum in new Internet applications and services such as online video, networked gaming, file-sharing, social network and cloud computing. Many of these have become an integral part of people's daily life. As a result, network traffic over Internet Protocol (IP) is growing at an accelerated pace, which in turn poses new challenges and requirements for IP router designs.

IP routers are the backbone devices of Internet. A router connects multiple networks via its I/O ports. The incoming data of IP routers process are organized as packets. Upon arrival from an input port of a router, a packet is processed and then forwarded to a desired output port. Typically, the data processing consisting of a series of operations on the header field of every packet is performed by dedicated hardware and/or procesors, whereas the actual forwarding is usually achieved by a switch fabric. In this work, we primarily focus on packet processing part, but leave packet switching for future work.

Today's IP routers are under a unique set of design specifications due to the exponentially increasing bandwidth requirements and the fast-changing network protocols and applications. As a matter of fact, none of current router solutions could simultaneously meet the often-conflicting requirements on performance, programmability, power and cost [1]. Traditionally, routers depend on application specific or "domain-specific" ICs to deliver the highest performance. With the skyrocketing fabrication cost due to the aggressive scaling of the semiconductor process, however, it is becoming infeasible to develop a cost-efficient IC solution targeting a relatively smaller market. In addition, the related customer base is too small to attract sufficient software support. The above observation is exemplified by the fact that Intel recently closed its network processor product line [2]. On the other hand, software routers utilize general-purpose processors to offer the best programmability and flexibility. Such solutions are cost-efficient and supported by powerful software development tools. Nevertheless, the major drawback of software routers is that they can only deliver a throughput of 1-3Gbps, which is considerably lower than the required throughput of 40Gbps - 92Tbps for core networking equipments [3].

In the recent years, Graphics Processing Units (GPUs) are emerging as a new general-purpose computing platform that offers both high performance and strong programmability. The massive scale of GPU user community also guarantees sufficient support for software development. It is thus appealing to use GPU to address the above dilemma of router designs. A few recent works already demonstrate the potential of GPUs for high performance routing processing [4], [5]. Nevertheless, IP routers have to meet stringent quality-of-service (QoS) requirements such as packet latency and latency variance, while so far GPU architectures do not have direct control on such QoS metrics. In this work, we propose a solution to the above problem by augmenting a mature GPU ar-

- *Y. Zhu is with Department of Electrical and Computer Engineering, the University of Texas at Austin, Austin, TX 78741. E-mail: yuhao.zhu@mail.utexas.edu.*
- *Y. Chen is with the Institute of Microelectronics, Tsinghua University, Beijing 100084. E-mail: yb08@mails.tsinghua.edu.cn.*
- *Y.Deng is with the Institute of Microelectronics, Tsinghua University, Beijing 100084. E-mail: dengyd@tsinghua.edu.cn.*

chitecture and its programming tools. We also present an analytical performance model of the proposed microarchitecture. The model enables us to quickly estimate the packet processing performance under a given network traffic pattern. It also provides insight for designing massively parallel packet processing engines. In summary, the major contributions of this paper are as follows:

- We developed an integrated heterogeneous CPU/GPU microarchitecture, Hermes, for massively parallel packet processing. The CPU and GPU are closlely coupled with a simple yet effective interface. By sharing a common memory hierarchy, the communication overhead between CPU and GPU is minimized.
- To the best of the authors' knowledge, this is the first work to introduce QoS management mechanism to GPU-like massively parallel architectures.
- We propose a complete analytical performance model for Hermes and other similar massively parallel architecture. Our model enables fast throughput and latency estimations so that designers can efficiently explore the solution space.

The rest of the paper is structured as follows. Section 2 reviews the background of this work. The details of hardware and software designs of Hermes are introduced in Section 3. Section 4 presents a thorough performance evaluation of Hermes. Section 5 discusses key design considerations and constructs analytical models for Hermes. Related works are reviewed in Section 6. Section 7 concludes the paper and outlines important future research directions.

2 Background and Motivations

2.1 IP Routing

The essential task of an IP router is to determine the destination port (i.e., connection to different networks) of each individual packet in the incoming Internet traffic. Such a decision is made according to the header information of each packet. Besides the above forwarding operation, an IP router also performs a series of actions for flow control and bookkeeping. A typical processing flow for a given packet consists of such steps: 1) checking the IP header (CheckIPHeader) to verify the validity of an IP packet, 2) classifying the packets (Classifier) to identify flows and filter data traffic etc., 3) looking up a routing table (RTL) to determine the outgoing port, which is in fact the primary task of an IP router, 4) decrementing the time-to-live value (DecTTL), and 5) fragmenting the packet (Fragmentation) to small ones in order to fit the Maximal Transfer Unit of a network [46]. In addition, the ever-demanding requirements for intrusion detection have made deep packet inspection (DPI) [7] a regular task on the critical path of packet processing. Throughout the paper, we will use the above processing pipeline to evaluate our system.

2.2 Current IP Router Solutions

Today's commercial and academic IP routers can be classified into three categories: hardware routers, software routers and programmable network processors (NPs) [6]. The selection of a specific solution depends on complex tradeoffs among such metrics as performance, programmability, power budget, area efficiency, scalability, and marketing considerations. In particular, router performance is measured in terms of quality-of-service (QoS). We will further discuss the QoS issues of a router in Section 4.

Hardware routers were once the most commonly chosen routing solution. They depend on customized hardware, i.e., ASICs to deliver the highest performance with the least power/area overhead. On the other hand, hardware routers suffer from the long design turnaround time, poor scalability, and inferior programmability. Such limitations gradually made ASIC based solutions out of the mainstream. The adoption of FPGA based solutions mitigates some of the problems, but still cannot offer sufficient programmability.

In contrast, software routers implement all packet processing applications as programs running on commodity multi-core platforms or clusters/server-farms. Therefore, they offer the highest flexibility because the programmability makes it straightforward to reflect arbitrarily any changes in network configurations and protocols. The advantage of such an approach is even more significant when considering the scales of market and customer base. In fact, general-purpose processors are targeting a much larger market and thus supported with more mature operating systems and development tools. Both the openness in hardware architectures and software tools make software routers desirable for today's constantly changing network applications and services. However, it is extremely challenging for pure software implementations to deliver sufficient computing power required by high performance networks. Therefore, such routers are usually used for routing services in relatively small networks.

In the middle of the solution spectrum is the network processors (NPs) based IP routers. NPs are designed to hit a balance among performance, cost efficiency, and flexibility by integrating many general-purpose processing engines (PEs) optimized for data level parallelism (DLP) [1]or task level parallelism (TLP),[2] as well as a set of special-purpose coprocessors that are either hardwired or configurable for packet processing [8]. The PEs and the coprocessors are coordinated by a task scheduler. However, a clear downside of network processors is that so far an effective programming model has not been constructed due to the limited size of market and customer base [9]. As a result, it takes great efforts to develop efficient applications that fully unleash the potential computing power of NPs. Meanwhile, the small volume of NPs also leads to prohibitive per chip cost. As a matter of fact, the above two problems already force some top NP vendors to

[1] PEs are organized as parallel modules, i.e., one packet is processed in one PE

[2] PEs are organized as pipelined modules, i.e., packet processing is divided into multiple tasks and one or more stages are responsible for one task

resort to multi-core based router solutions [10].

2.3 GPU Systems

Recently GPU based general-purpose computing has become an important trend [11]. Under such a computing paradigm, GPUs are regarded as the coprocessors or accelerators of the CPU to exploit the massive parallelism presented in computing-intensive applications. In this subsection we discuss why this heterogeneous architecture offers significant potential to be a packet processor from both hardware and software points of view.

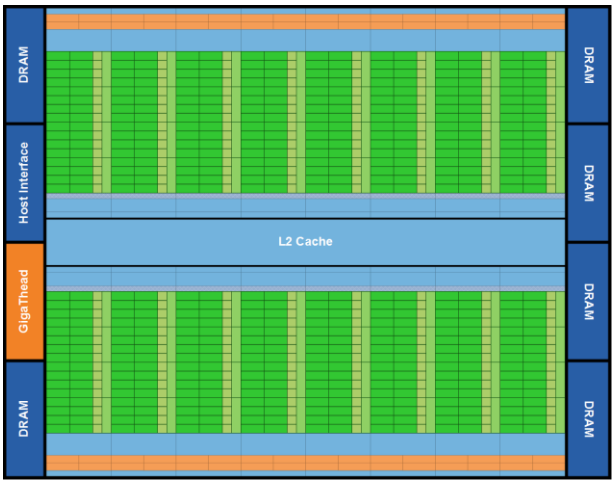

Fig. 1. Architecture of NVidia Fermi GPU

Figure 1 illustrates the high level organization of a 32-wide [3] SIMD GPU architecture (i.e., NVIDIA's Fermi GPU). It contains 16 shader cores (SC), each equipped with 32 scalar processors (SP). When running a typical GPU program, a massive number of threads are grouped into 32-wide thread warps and are then executed on different shader cores. A warp of threads adopt a single instruction multiple data (SIMD) model, since they share the same instruction fetch/issue unit and follow an identical instruction schedule. In addition, modern GPUs have hardware support for intra-warp divergence, and thus it is not necessary to comply with the restrictions imposed by traditional SIMD architectures in which threads in a warp must follow exactly the same control flow. Upon divergence, branches are taken one after another and reconverge at a given node of the underlying control flow graph [12]. From this perspective, the GPU execution model can be regarded as single program multiple data (SPMD) or single instruction multiple thread (SIMT) [13] as defined in NVidia's terminology.

The nature of packet-based network processing does not require communication between two processing elements for two different packets. Accordingly, processing procedures can be replicated on scalar processors, which then behave like processing engines in traditional network processors. Clearly, the execution model of shader cores is suitable for massively parallel packet processing.

Modern GPUs generally employ a fine-grained multi-threading [14] mechanism as well as a flexible memory hierarchy to hide the memory access latency. Both features are essential for packet processing applications. The largest GPU memory resource is the global memory shared by all the shader cores. Although it takes hundreds of cycles for a complete access, L1 and L2 caches are installed to reduce the waiting time. Moreover, the fine-grained multithreading execution mechanism interleaves the operations of different warps. In addition, when a currently active warp is waiting for a memory access, it can be suspended and another ready-to-run warp can be activated. Shader cores on a GPU generally have its own texture and constant caches, which are ideal for fast indexing of constant and regular data like routing tables. Besides, within each shader core, there is a software managed shared memory which allow the same accessing speed as the SP registers as long as no bank conflict. Although it is designed for the inter-thread communication within a shader core, the shared memory can serve as an extension of the abovementioned hardware cache to facilitate the memory accessing performance in packet processing applications.

Finally, GPU programming has been made much easier than the hard-to-learn programming model of network processors. The release and wide acceptance of GPU development tool-chains including programming models like CUDA [15] and CTM [16] as well as runtime and debugging systems allow developers to implement their applications onto GPU platforms in a fashion similar to that of conventional C/C++ development. In addition, there is already a large GPU programming community. Therefore, the GPU software eco-system has a strong potential to meet the ever-growing requirements for rapidly deploying new network protocols and services.

2.4 Limitations of GPU based Software Routers

Two recent works [4] and [5] already proved the potential of GPUs for packet processing. By implementing the router application as CUDA programs, a GPU based software router solution outperforms a CPU baseline router by a factor of up to 30X. However, two main problems hinder a wider adoption of heterogeneous CPU/GPU systems for software routing applications.

First, the communication mechanism between CPU and GPU seriously degrades system throughput. In a typical packet processing scenario, the CPU first transfers packets to GPU for routing processing after initialization. After the routing processing is finished, GPU initiates a data transmission again to move the data back to CPU. Under certain circumstances, packets data have to be transported between CPU and GPU back and forth for multiple times. However, the CPU-GPU communication is through a PCI Express (PCIe) bus [18] with a peak bandwidth of only 16GB/s. Compared with the over 100GB/s bandwidth between GPU and its memory, the PCIe bus is clearly a bottleneck., not to mention that we also need extrea memroy copies.

The situation is illustrated in reference (as shown in Figure 5) [4]. It is reported that the overall throughput of GPU processing can be more than 30 times higher than

[3] Typical NVidia's GPUs are 8-wide SIMD machines. However, in the most recent Fermi architecture, a 32-wide SIMD architecture has been employed. We adopt this new organization to avoid the complexity of super-pipelining.

that of CPU without counting the CPU-GPU transfer cost. When considering the data transfer overhead, however, the speed-up degrades to 5X.

Second, GPU's batch processing model hurts the worst-case delay. In principle, GPUs are designed as throughput oriented processors that emphasize average thread throughput. They employ hundreds of execution units that can execute different threads in parallel to attain a very high throughput. In order to utilize the execution resources, GPUs' programming model requires a sufficient number of threads to be available before they can be instantiated on GPU for further computation. In the context of packet processing, previous works typically assign one packet to one GPU thread. Therefore, early-arriving packets have to be waiting on the CPU side before an enough number of packets are accumulated. Such a batched processing fashion may lead to a long latency for some packets. This negative effect conflicts with the QoS requirements and is many times intolerable for delay sensitive applications such as online video and teleconferencing. Here, the key insight we gained from previous works is that it is essential to deliver a balanced solution for overall QoS metrics.

In summary, the above analysis forms the foundation of this work on enhancing the GPU microarchitecture for network processing. Here we have to follow two rules of thumb: 1) the overhead of data transfer must be minimized; and 2) it is essential to hit a balance between system throughput and average packet delay.

3 HERMES SYSTEM

3.1 System Overview

Figure 2 presents a high level overview of the Hermes system. In the Hermes system, we propose two microarchitectural innovations on current GPU microarchitectures for network processing. The first one is to integrate CPU and GPU with a shared memory space to minimize the communication overhead and thus improve the system throughput. The other is to employ an adaptive warp (packet) issuing mechanism to achieve better average packet processing delay. In this section we discuss these two microarchitectural features as well as the corresponding software extensions.

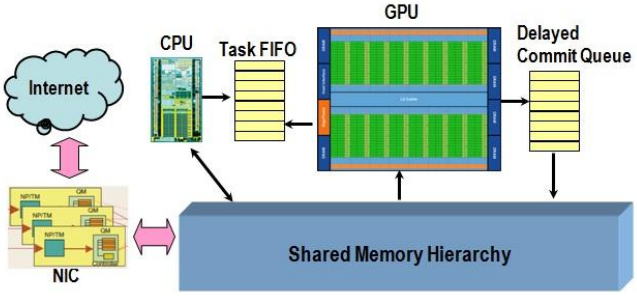

Fig. 2. Hermes overview

3.2 CPU-GPU Integration Via Shared Memory

As explained in Section 2.4, improving the bandwidth between CPU and GPU is critical for such heterogeneous systems to be pratical in use. A shared-memory architecture inherently resolves this problem. Fortunately, with the rapid-growing integration capacity made available by the advancement of semiconductor process, it is now feasible to deploy closely-coupled CPU and GPU cores on a single chip. Pangaea [47] is one of early explorations along this direction. The Hermes architecture proposed in this work follows a similar philosophy. Rather than using separated memory spaces, CPU and shader cores in Hermes system share the same memory hierarchy so as to remove the extra memory copies between CPU and GPU. The GPU still works as a coprocessor of CPU, as we want to minimize the changes to the current CPU/GPU architecture and computing paradigm. In other words, our design philosophy is to develop an enhanced architecture but keep compatibility with the current GPU programming model.

Accodingly, in the Hermes microarchitecture, data communication between CPU and shader cores is through the shared memory hierarchy rather than a PCIe bus. The overall execution flow remains to be the same as a classical heterogeneous CPU/GPU platform, e.g., CUDA. The CPU is responsible for creating and initializing data structures according to packet processing applications. This process can be regarded as a system configuration stage. When a given number of input packets are available, the CPU stores packet data into the shared-memory and then launches a kernel, in which one thread is associated with one packet. Shader cores in GPU fetch data from the shared memory and perform the corresponding 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 destination ports.

Another implication of maintaining compatibility with a current GPU programming model is that Hermes does not have the race condition problem that is typical in shared-memory architectures. In fact, the data accessing by CPU and GPU are independent under the CUDA programming model. In other words, the CPU and GPU operations on an individual packet are inherently mutually exclusive. Nevertheless, it must be noted that the out-of-order commit of finished packets may introduce consistency problems, which will be further discussed in the next sub-section.

An important function of shared-memory is to serve as a large packet buffer to avoid the canonical buffer sizing problem ([20], [21], [22]). According to [23], the optimal size of router buffer should be determined by the "Bandwidth-Delay Product" (BDP) as a rule-of-thumb. In a typical network environment, such a guideline mandates a 1.25GB buffer size [24], which is impractical for traditional router designs. On the other hand, routers using smaller buffers suffer from a high packet loss rate [25]. Accordingly, 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 optimal packet availability.

The shared-memory architecture significantly reduces the overhead of data communication between CPU and its coprocessors. However, one remaining question is

how the CPU controls its coprocessors in such an integrated system, especially when dedicated for packet processing applications. The following section resolves this question by proposing an adaptive warp issuing mechanism.

3.3 Adaptive Warp Issuing

In this work, a key enhancement to current GPU microarchitectures is a warp issuing mechanism, which organizes data-parallel tasks and then assigns them onto shader cores. When a given shader core receives warps, its warp scheduler[4] will then determine when a warp should be activated or suspended. In a traditional CPU/GPU system, 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[5] size. However, due to the streaming nature of packet processing and the requirement for real time processing (as explained in Section 2.4), it is not affordable to wait for an enough number of warps. Therefore, we propose 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.

The key structure that enables adaptive warp issuing is a simple task FIFO as illustrated in Figure 2. The packets arrive at a router via NICs and then are DMA'ed to the shared memory. CPU is keeping track of the number of arrived packets and notifies the GPU to fetch packets for processing by putting the number of available packets in the task FIFO.

Specifically, CPU creates a new FIFO entry with the value of the number of packets ready for further processing as soon as it decides it is appropriate to report the availability of packets and the task FIFO is not full. Meanwhile, the GPU is consistently monitoring the FIFO and making decisions on fetching a proper number of packets. 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.

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 a tradeoff has to be made. On the one hand, transferring a packet from NIC to the shared memory involves a performance overhead (excluding DMA data copy) such as reading and updating the related buffer descriptors. The corresponding extra bus transactions may be unaffordable [26]. In addition, updating the task FIFO too frequently also complicates GPU fetching due to the restriction of finest fetching granularity mentioned before. Therefore, it is beneficial to minimize system bus transactions and update the task FIFO in a relatively coarser granularity. On the other hand, too large an interval between two consecutive updates increases average packet delay and should be avoided. Moreover, setting a lower bound on transfer granularity, in the worst case, would result in a timeout problem, which would delay the processing of some packets. If such a timeout really happens, the NIC logic and corresponding system 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 was taken by [26]. For simplification, the interval is chosen to be double warp arriving time, although an adaptive estimation based on the packet arriving rate would potentially be better. Upon finishing one transaction of data transfer (from NIC to system memory), the CPU notifies GPU through updating the FIFO. In our experiments, we found that such a configuration generally is able to guarantee timely packet processing.

Newly created warps are put in a warp pool, waiting for issuing. A round-robin issuing strategy is employed to evenly distribute the workload among each shader core. In attempting to achieve better load balancing, we also implemented a more precise strategy that tracks the occupancy of each shader core and then always issues warps to the least "hot" one. However, given a uniform packet traffic pattern, the impact delivers negligible performance gain but rather incurs a high hardware overhead.

Clearly, it is still important to keep track of the availability of hardware resource that eventually restricts the maximum number of concurrently active warps. Upon reaching that limit, the warp issuing should be paused until new warp slots are available. However, we find it possible to explicitly control that upper bound in order to achieve a better QoS. The details will be discussed in Section 5.

3.3.1 In-Order Warp Commit

Owing to 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 complete processing with any order. Some protocols such as TCP do not enforce the order of packets processing and committing since TCP header includes extra areas to enable retransmission of lost packets and reassembly of out-of-order packets into the correct sequence. However, others like UDP do require in-order processing of packets [27]. Therefore, if GPU commits packets to CPU (by writing the data back to the shared memory) in their finishing order, CPU and GPU may have an inconsistent view of packets status. To maintain the so called "packet consistency" over all protocols, it is mandatory to keep the packet commitment order the same as the arriving order. In network processors, a complicated task scheduler is responsible for this purpose

[4] A warp scheduler chooses data-ready warps to be fetched for execution in a multiplexing manner.

[5] Shared memory here indicates the memory storage installed in the shader core and shared by scalar processors. It is different from the "shared memory" in the termoinology "shared memory architecture".

[28], but our solution only requires a simple Delay Commit Queue (DCQ).

The key idea is to allow out-of-order warps execution but still enforce in-order commit. This resembles the Reorder Buffer (ROB) [29] in a processor with hardware-enabled speculation.

As illustrated in Figure 3, the DCQ holds the IDs of those warps that have been finished but not committed yet. Every time a warp is issued onto one shader core, the status of DCQ is checked. If it is not full, a new entry is allocated. This one-on-one mapping between warp ID to 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 unique id. Only could a warp be committed when all warps arrived earlier have been finished. The checking of warp status involves a traversal from the DCQ to the entry for the current warp. Since the number of warps is relatively small, the traversal can be implemented efficiently in hardware. Once a warp commits, its entry in DCQ is reclaimed.

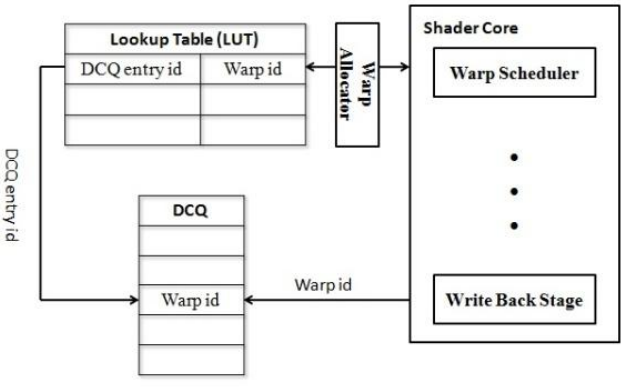

Fig. 3. Implementation of Delay Commit Queue (DCQ) with a Lookup Table (LUT)

Note that in this paper we always assume no dependency existing between any two packets, and therefore we can execute and commit them in the batch mode. Some previous work [48] explored a more general case where inter-packet dependencies do exist, which complicates the ordering issue. However, such dependencies make GPU implementations inherently unsuitable, and thus are beyond the scope of this paper.

3.3.2 Hardware Implementation and Cost Estimation

As compared with original GPU hardware, Hermes requires three additional memory components, the 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. We tested throughout all our benchmark applications, and found that a size of 1K entries for both task FIFO and DCQ suffice for typical packet traces. In fact, the circuit structure of task FIFO and DCQ are relative simply, enabling a much bigger implementation to minimize the chance of stall. Since task FIFO stores the number of available packets, its entries can be set with a size of one integer, i.e. 32 bits. Similarly, DCQ records the warp index and thus the size of its entries must be smaller than the total number of maximally allowed concurrent warp (MCW) over all shader cores. In our work, the number of MCW 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 can be safely chosen as equal 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 indentify 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 4.0 [30] to estimate the area cost of these three hardware add-ons. According to its SRAM model, task FIFO and DCQ cost 0.053mm^2 and 0.013mm^2 respectively, while 8 DCQ-Warp LUTs take 0.006mm^2 in total. As compared to the total area of one GPU chip, the hardware overhead is next to negligible.

3.4 API Modifications

Hermes programming model is based on CUDA [15]. It introduces 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 on top of CUDA, in the future it 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.

Hermes does not need the concept of of Cooperative Thread Array (CTA) but directly organize and schedule threads in warps. In traditional CUDA programming praradigm, threads are typically grouped into CTAs (with each usually contains many warps) on the CPU side before transferred to GPU. The concept of CTA is proposed and designed in favor of batch processing and provides a way for inter-thread sharing. However, adaptive warp issuing mechanism breaks such batching processing mode by enabling a finer-grained communication, typically in warps, between CPU and GPU, effectively abandoning the concept of CTA, not to mention that we always assume no dependency and sharing among packets.

An implication of the above decision is that we are now not able to computer 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 necessary information needed to program GPU kernels.

4 Experimental Evaluations

4.1 Methodology

In this work, we use GPGPU-Sim [31], which is a cycle-accurate GPU microarchitecture simulator that supports

CUDA programs, to evaluate our modifications. The GPU microarchitectural configurations used in this work are presented in Table 1.

It is worth noting that, in the current implementation of GPGPU-Sim, the host side CUDA codes run on a normal CPU (host in CUDA terminology), 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 the shared-memory architecture in terms of the overhead of PCIe transfers, which clearly dominate in the total overhead of the traditional data communication between CPU and GPU.

To evaluate our proposed architecture, we implemented a complete CUDA-enabled software router, which covers all the tasks as declared in Section 2. For DPI, we employed a bloom filter based algorithm [32] that is amenable for GPU implementation. The string rule sets were taken from Snort [33], while the replay traces of network traffic were extracted by Tcpreplay [34]. In the case of routing table lookup, packet traces are retrieved from RIS [35]. Packet classification implements the basic linear search algorithm and uses ClassBench [36] as the benchmark. The other three applications (checking IP Header, decrementing TTL, and IF fragmentation) are adapted from RouterBench [37], and tested under WIDE traffic traces [38].

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

Since Hermes is targeting the IP routing applications, Quality-of-Service (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 [39], [40]. 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 interval between the time it enters the router and the time it is ready for further forwarding. The delay metric consists of two components, queuing delay and serving 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. The waiting time is the queuing delay. The service delay is the time for a packet to receive complete processing by a shader core.

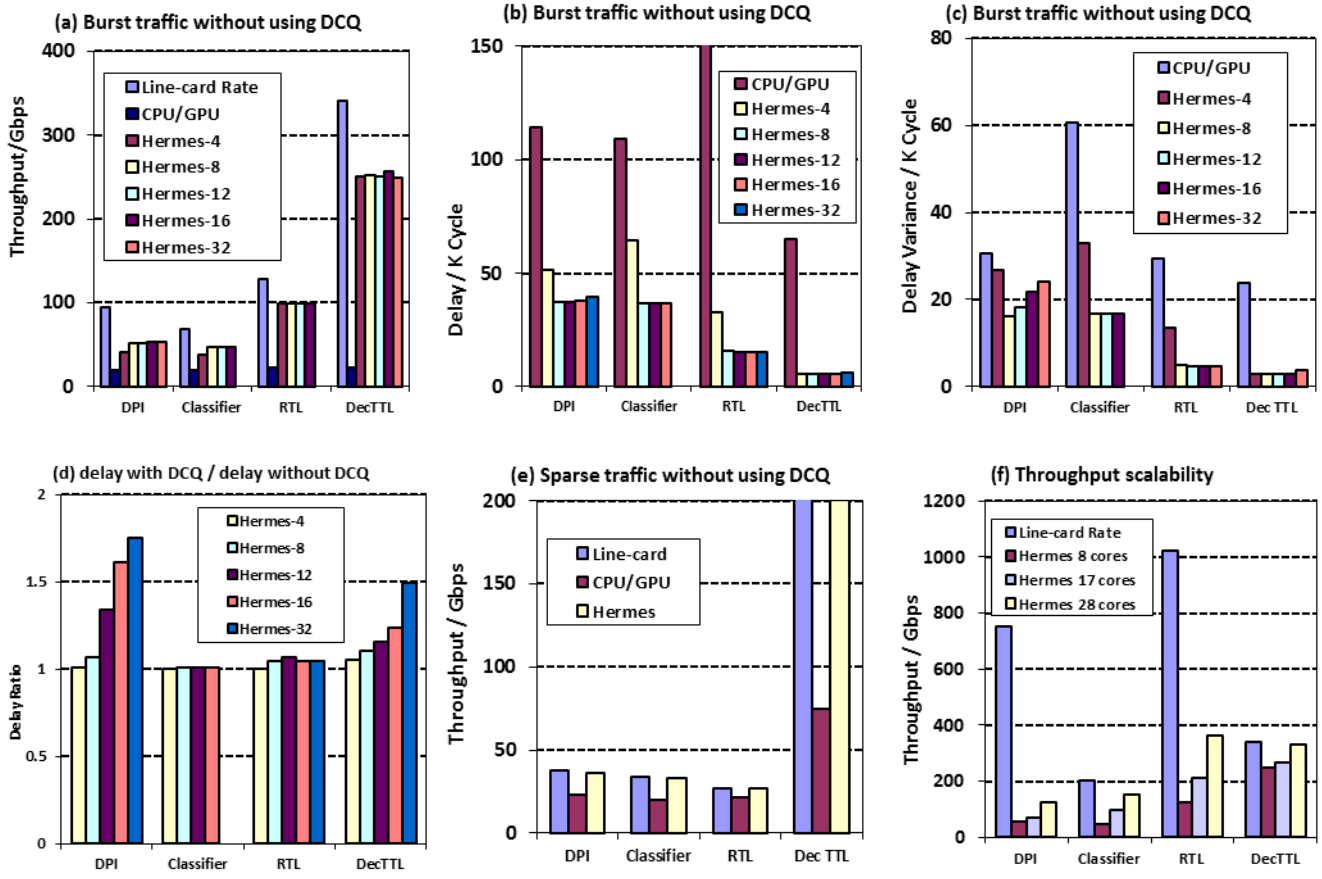

Fig. 4. QoS metrics (a) throughput under burst traffic, (b) delay under burst traffic, (c) delay variance under burst traffic, (d) delay ratio, (e) throughput under sparse traffic, and (f) throughput scaling with # shader cores.

Note that the time spent in the DCQ is also included in the serving delay. The delay disparity of different packets is defined as delay variance. We use the interquartile range to measure it.

4.2 Results

According to our profiling, DPI, packet classification and routing table lookup together consume nearly 90% of total processing time. The remaining three applications, CheckIPHeader, DecTTL, and Fragmentation, are much less demanding. In addition, the latter three applications have almost identical behavior processing patterns. Therefore, we use DecTTL as a representative to explain the results of the latter 3 applications.

Figure 4 shows the three QoS metrics of the four benchmark applications. The number of Maximally-allowed Concurrent Warps (MCW) and 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 4(a) to 4(d). A sparse traffic is applied in 4(e). Both bust and sparse traffics are used in 4(f). Each application has their own line-card rates provided by traffic traces, as showed in the leftmost column of four column-sets in 4(a), 4(e) and 4(f).

Figure 4(a) compares a traditional CPU/GPU system against different configurations of Hermes. Note that the overall processing time of CPU/GPU system consists of three components, packet-waiting overhead (the time it takes 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 somehow violates the throughput-oriented design philosophy of GPU. In Section 5.2, we develop a model to estimate the maximal throughput of Hermes.

Hermes can deliver network packets with a much smaller delay than a traditional CPU/GPU system as showed in 4(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 in GPU, the waiting overhead at CPU side contributes to a non-negligible part of total delay. Therefore, delay improvement is more significant for these two applications, since Hermes could overlap CPU side waiting overhead with GPU processing. Comparing different configurations of Hermes, Hermes-4 (i.e., #MCW equals 4) always performs the worst. Classifier and RTL do not present significant difference for other three configurations. For DPI and DecTTL, Hermes-32 performs slightly worse than other two configurations. On average, the best case of Hermes can reduce packet delay by 81.2%. We further discuss the delay in Section 5.3.

As showed in 4(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, however, Figure 4(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 (as defined in the next Section) of Hermes shader cores. As illustrated in Figure 4(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 under such a situation, a CPU/GPU system is still unable to deliver the packets at their arriving rate.

Finally, 4(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 rather satisfactory. Note that in DecTTL, the scaling factor does not completely follow the increasing number of shader cores. It is because the arriving packet rate is too sparse for a Hermes with 28 shader cores and thus the computing resources are not fully utilized, as justified by the fact that in DecTTL the throughput of Hermes-28 is approximately equal to the line-card rate.

5 Analytical QoS Modeling for Hermes

5.1 Discussions

The Hermes architecture is flexible and scalable in the sense that its high-level microarchitectural parameters such as the number of shader cores and the number of maximally allowed concurrent warps can be determined according to the incoming Internet traffic. Such customizability is especially useful when the network traffic follows certain fixed patterns (Poisson distribution for incoming phone calls is a well-known example). The flexibility and scalability of Hermes suggests that it can be instantized with a specific configuration to meet the requirements of a particular networking environment. We hereby present a high-fidelity analytical model for fast microarchitectural explorations to identify an ideal configuration. Note that our model captures the generic patterns of data parallel streaming applications, although it is developed from packet processing applications. Accordingly, the models can also be applied to other application domains to provide key design insight.

In the remaining of this section, we first identify the raw computing capacity of GPU by introducing the concept of computing rate. We then construct an analytical model for the maximal system throughput as well as av-

erage packet delay in a Hermes system.

5.2 Computing Rate and Throughput Model

The concept of computing rate for a shader core is introduced to measure the potential performance for an application. Before looking into the details, we first define computing rate (CR) as in (1).

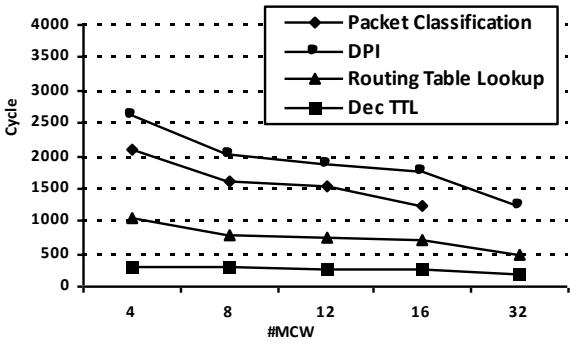

Fig. 5. Computing rates of benchmark applications

Assuming all the shader cores in the GPU are identical, the computing rate for a shader core as a function of MCW is defined as the average time interval between two finished warps running on the same shader given all #MCW warps are active. It is equivalently the average processing time for one warp as calculated by (1). Obviously, smaller computing rate indicates more powerful computing capacity.

Through profiling, we derived the computing rate values for our benchmarking application in Figure 5. Clearly, as maximally allowed number of concurrent warps becomes larger, hardware resources can be more efficiently used, thus the computing rate becomes higher. Owing to the relatively less-demanding computations in DecTTL, its computing rate curve remains almost constant.

Clearly, when the packet arriving rate, i.e., the line-card rate exceeds the computing rate, more and more packets would be buffered (queued) before they can be served. In this case, the system throughput would no longer scale with the line-card rate. Accordingly, we can compute the system maximal throughput, which is achieved when the line-card rate equals computing rate. The results are listed in Table 2. We set the number of shader core (#SC) to be 8. Clearly, throughput reported here would scale with #SC as showed in Section 4.2.

TABLE 2
MAXIMAL THROUGHPUT (GBPS) FOR HERMES

#MCW	Packet classification	DPI	Routing table lookup	Dec TTL
4	38.7	40.87	77.4	250.4
8	48.4	52.71	99.6	258.2
12	48.4	55.59	100.0	240.6
16	48.5	55.24	100.1	254.2
32	NA	56.95	121.3	248.7

$$CR_{\#MCW} = \frac{\#shader\ cores \times \#total\ processed\ cycle}{\#total\ processed\ warps} \tag{1}$$

$$\begin{aligned} D_m &= Queuing\ Delay + Service\ Delay \\ &= [(m-1)\times \#MCW \times CR_{\#MCW} - \sum_{i=1}^{(m-1)\times \#MCW} C_i\ ] + \#MCW \times CR_{\#MCW} \end{aligned} \tag{2}$$

$$\begin{aligned} \overline{D} &= \frac{1}{m} \times \sum_{i=1}^{m} D_i = \#MCW \times CR_{\#MCW} + \frac{(m-1)\times \#MCW \times CR_{\#MCW}}{2} - \frac{1}{m} \times \sum_{u=0}^{m-1} \sum_{i=1}^{u\times \#MCW} C_i \\ &= \#MCW \times CR_{\#MCW} + \frac{(m-1)\times \#MCW \times (CR_{\#MCW} - \overline{C})}{2} \end{aligned} \tag{3}$$

$$\overline{D} = \#MCW_X \times CR_{\#MCWs} = \overline{C} \tag{4}$$

$$\overline{D} = \frac{\int_{t1}^{t2} \frac{dt}{C(t)} \times \#MCW \times C(t)}{\int_{t1}^{t2} \frac{dt}{C(t)}} \tag{5}$$

$$\int_{t_{s1}}^{t_r} \frac{dt}{C(t)} \times CR_{\#MCW} = t_r - t_{s1} \tag{6}$$

$$m = \frac{1}{\#MCW} \int_{t_{s1}}^{t_r} \frac{dt}{C(t)} \tag{7}$$

$$\overline{D(t_{s1}, t_r)} = \#MCW \times CR_{\#MCW} + \frac{1}{m} \times \sum_{u=0}^{m-1} \sum_{i=1}^{u \times \#MCW} C_i \tag{8}$$

$$\overline{D(t_r, t_{s2})} = \frac{\int_r^{s2} \frac{dt}{C(t)} \times \#MCW \times C(t)}{\int_r^{s2} \frac{dt}{C(t)}} \tag{9}$$

5.3 Delay Model

As compared to the overall throughput, real-time network applications are more sensitive to packet delay. Therefore, architectural design decisions should be carefully made to avoid deteriorating the average delay. Here we take a first step to discover its dependency with the number of maximally allowed concurrent warps.

Comparing Figures 4(b) and 4(e), it can be seen that the delay with DCQ is still much better than that of a traditional CPU/GPU architecture even in the worst case. We thus construct our model without considering DCQ for simplicity. In fact, the overall trend would not change even when taking DCQ into account, as justified by simply multiplying the corresponding columns in 4(b) and 4(e).

We define an integer series C, which describes the time interval between two consecutively arrived packet warps. $\overline{C}$ can thus be regarded as the average interval between two arrivals. Statistically, $CR_{\#MCW} = \overline{C}$ is the threshold condition that does not require packet buffering, i.e., sparse traffic (as opposed to burst traffic). We first discuss these two traffics separately, and then establish a combined model for a mixed traffic.

5.3.1 Burst Traffic Model

Given a bursty traffic, the delay for one warp consists of queuing delay and service delay. The service delay roughly equals #MCW × $CR_{\#MCW}$ since now all #MCW warps in a shader core are active. The queuing delay can be computed by subtracting the time when a packet arrives from the time it is serviced (both measured in cycles). We organize arriving packet warps into groups with a size of #MCW. Due to the bursty nature, the delay of the first warp in a group can be regarded as an approximation of the average delay for all warps in this group. Without loss of generality, in the m^{th} group every warp has the average delay described by (2). Thus, we can calculate the average delay for all m groups of warp as (3).

Figure 6 presents the measured and estimated time for the four benchmark applications. We use a long traffic trace to reduce errors. The results show the average geome-

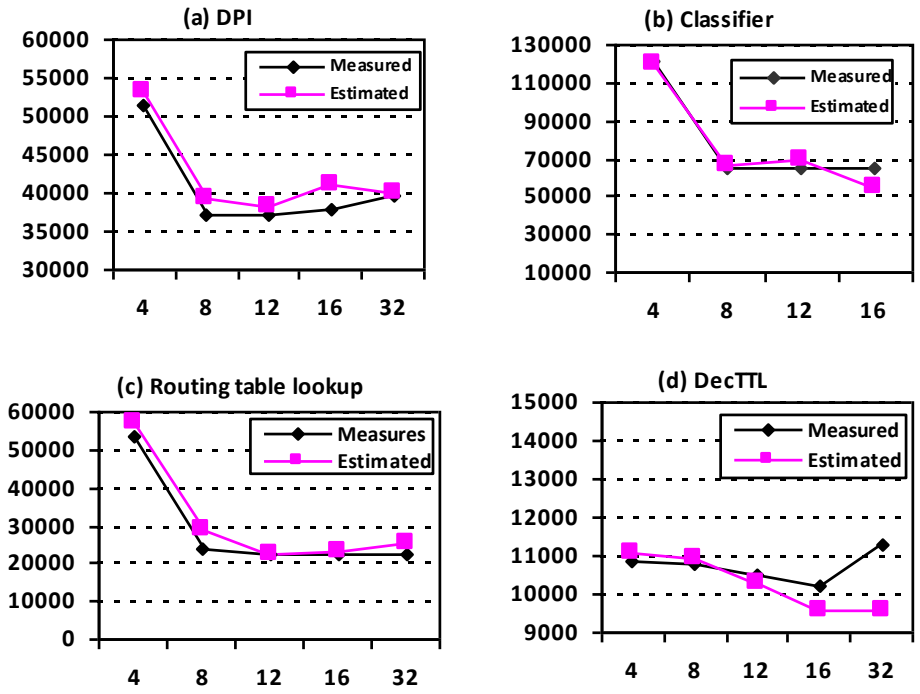

Fig. 6. Effect of Burst Model

tric mean of error is 7.8%.

5.3.2 Sparse Traffic Model

In this case, we have $\overline{C} > CR_{\#MCW}$. Therefore, according to the curves in Figure 5, we can find a specific $\#MCW_s$, such that $CR_{\#MCWs} = \overline{C}$. Statistically, this $\#MCW_s$ is the average number of active warps in the shader under current traffic. In other words, we claim that given a particular uniform sparse traffic and #MCW setting, there are only $\#MCW_s$ warps active on average and every $CR_{\#MCWs}$ cycle would one warp be finished. As a result, under such a uniform sparse traffic, the average delay can be described as (4). In Figure 7, we check the accuracy of above model by replaying the traffic in Figure 4(e). On average, the error rate is 9.1%.

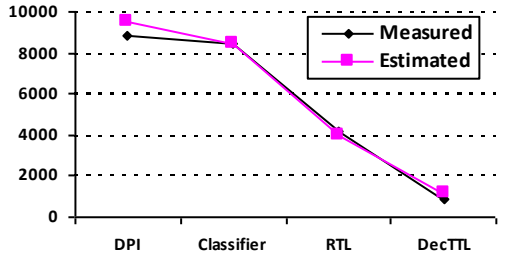

Figure 7. Effect of Sparse Model

More generally, if the sparse traffic is non-uniform, we cannot use $\overline{C}$ for approximation. Instead, C has to be treated as a function of time, i.e., C(t). Therefore, we have to integrate (1) from t_1 to t_2. In this case, the average delay should be (5).

5.3.3 A Unified Model

In the last two sub-sections, we developed delay models for bursty traffic and sparse traffic, respectively. The above two patterns can be considered as two extreme cases of network traffic. For an arbitrary traffic pattern represented as the distribution of the line-card rate over a given time, a generic model is desired for estimation.

Here we need to define (1) the accumulation starting point as the time when packets start to accumulate in the buffer, and (2) the accumulation resolve point as the time when all the packets in the buffer are right issued (i.e., accumulation in the buffer is resolved). It is obvious that between two accumulation starting points T_{s1} and T_{s2}, there is one and only one accumulation resolve point Tr. The relationship should satisfy (6).

Intuitively, between T_{s1} and T_r, the traffic can be regarded as bursty, while the traffic is sparse between T_r and T_{s2}. For the burst traffic in this case, we still divide incoming packets into groups of #MCW. The number of group follows (7). Therefore, we can use (3) to calculate the average delay between T_{s1} and T_r as in (8), and (6) to calculate average delay between T_r and T_{s2} in (9).

6 Related Work

There have been a couple of works focusing on leveraging GPU computing power for network packet processing. The work proposed in [17] explored the potential of GPUs to perform signature matching. The authors also identified key performance-hindering factors such as memory bottleneck and architectural restrictions, which are compatible with our modifications in this work. Mu et al. [4] performed the first work to implement GPU accelerated routing tasks such as routing table lookup and pattern match. Han et al. [5] implemented a complete GPU-based software router in. Both works demonstrate that GPU can accelerate packet processing by one order of magnitude. All the above works rely on the existing CPU and GPU model, while our work identified performance bottlenecks and developed architectural enhancements and corresponding API modifications.

Some recent IP router (packet processing) solutions are orthogonal to ours. RouterBricks [42] is a scalable software router based on a set of state-of-the-art, general-purpose servers running Click [43]. Although yielding high performance, the RB4 prototype is less cost-efficient than off-the-shelf GPU cards as the performance requirements scale. PLUG [44] is a complete solution including the tile-based architecture, programming model and runtime system to facilitate deployment of lookup modules in new network protocols. However, our solution relies on mature hardware and software architectures with minor extensions.

There also exist some analytical models for GPU in the literature. As compared architecture models such as [49], our model is a QoS model rather than an architectural model. Particularly, our model takes into account the adaptive warp issuing mechanism and correlates it with the QoS metrics. In addition, when coupled with power models such as [50], our model can also be used in low energy design for Hermes-like architectures.

Heterogeneous integrated architecture is gaining adoption in both academia and industry. Our work reported in this paper is inspired by Pangaea [47], which tightly couples an IA32 CPU with an Intel GMA GPU (without graphic legacies) on a chip multiprocessor. Our design differs from theirs by treating architectural level modifications as add-ons to mature hardware, while Pangaea requires more microarchitectural alterations. Both AMD and Intel recently announced their integration solutions. Intel's Clarkdale Core i3 and i5 combine CPU and GPU dies in one package. AMD's Fusion [19] builds such a hybrid architecture on a single piece of silicon, and the products will be shipped in this year. The above solutions can be directly employed as the underlying architecture for Hermes.

7 Conclusion and Future Work

Recent works proved the potential of GPUs for high speed packet processing. However, the lack of guarantee on QoS as well as the communication overhead between CPU and GPU turn out to be the major hurdles that hinder the deployment of GPUs in main-stream software router solutions. To overcome these limitations, we developed a novel closely-coupled CPU/GPU microarchitecture with a flexible scheduling mechanism that could adaptively maintain a balance between packet delay and overall throughput. A complete set of router applications has been implemented on this architecture. Experimental

results prove that the new GPU architecture meet stringent delay requirements, while at the same time maintain a high processing throughput. In addition, we also developed analytical models for both system throughput and packet delay. With such models, we are able to quickly identify an optimized Hermes configuration for QoS requirements in a particular network environment. 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 addition, this work provides a case study on customizing/extending GPU microarchitecture for a specific application domain.

In the future, we will continue our work in several directions. First, with Hermes serving as the packet processing engine, now the DMA transfer from NIC to main memory will become a bottleneck that limits the overall throughput. Hence, we are going to explore the possibility of a better communication mechanism between NIC and Hermes. Second, GPU memory system scheduling and instruction fetch are key to extract data level parallelism [41]. It is thus imperative to investigate the effectiveness of different memory scheduling and instruction fetching heuristics under the context of adaptive warp issuing mechanism. Third, we are also evaluating different CPU/GPU integration mechanisms other than shared-memory. For example, memory storages of CPU and GPU can still be integrated on one chip but with different address space, rather than a uniform one. It is claimed to be faster for CPU-GPU data transferring [19]. Finally, the Hermes architecture will greatly benefit through exploiting 3-D integration features [45]. Besides allowing a much high memory bandwidth, 3-D IC technology also enables more aggressive solutions such as seamlessly integrating accelerators like PLUG [44] with a Hermes processing engine.

REFERENCES

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

[2] R. Merritt, "Intel shifts network chip to startup," *EE Times*, http://www.eetimes.com/news/latest/showArticle.jhtml?articleID=202804472. 2007

[3] W. Eatherton, "The push of network processing to the top of pyramid," *Keynote Speech at ANCS*, 2005.

[4] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. Deng, and S. Zhang, "IP Routing Processing with Graphic Processors," *Proc. of DATE*, 2010.

[5] S. Han, K. Jang, K.S. Park, and S. Moon. "PacketShader: a GPU-Accelerated Software Router," *Proc. of SIGCOMM*, 2010.

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

[7] G. Varghese. "Network Algorithmics," Elsevier/Morgan Kaufmann. 2005.

[8] M. Peyravian, and J. Calvignac. "Fundamental architectural considerations for network processors," *International Journal of Computer and Telecommunications Networking*, 41(5), 2003.

[9] C. Kulkarni, M. Gries, C. Sauer, and K. Keutzer, "Programming challenges in network processor deployment," *Proc. of CASES*, 2003.

[10] Intel, "Packet Processing with Intel® Multi-Core Processors," *Inter Whitepaper*, 2008.

[11] D. Blythe. "Rise of the Graphics Processor," *Proc. of IEEE*, 96(5) 761–778, 2008.

[12] W. Wilson, L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," *Proc. of MICRO*, 2007.

[13] E. Lindholm, J. Nickolls, S.Oberman and J. Montrym. "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, 28(2): 39–55, 2008.

[14] M. R. Thistle, and B. J. Smith. "A processor architecture for Horizon," *Proc. of SC*, 1988.

[15] NVidia, "CUDA Programming Guide 2.3," 2009.

[16] J. Hensley. "AMD CTM overview," *International Conference on Computer Graphics and Interactive Techniques*, 2007.

[17] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan, "Evaluating GPUs for Network Packet Signature Matching," *Proc. of ISPASS*, 2009.

[18] PCI SIG, "PCI Express Base 2.0 specification," 2008.

[19] Nathan Brookwood, "AMD Fusion Family of APUs: Enabling a Superior, Immersive PC Experience," *AMD Fusion Whitepaper*, 2010.

[20] D. Wischik, and N. McKeown, "Buffer Sizes for Core Routers," *ACM SIGCOMM Comp. Communications Review*, 35(3), 2005.

[21] G. Raina, D. Towsley, and D. Wischik. "Control Theory for Buffer Sizing," *ACM SIGCOMM Comp Communications Review*, 35(3), 2005.

[22] M. Enachescu, Y. Ganjali, A. Goel, and N. McKeown. "Routers with Very Small Buffers," *ACM SIGCOMM Computer Communications Review*, 35(3), 2005.

[23] C. Villamizar, and C. Song. "High Performance TCP in ANSNet," *ACM SIGCOMM Comp. Communications Review*, 24(5), 1994.

[24] A. Vishwanath, V. Sivaraman, and M. Thottan. "Perspectives on Router Buffer Sizing: Recent Results and Open Problems," *ACM SIGCOMM Comp. Communications Review*, 39(2), 2009.

[25] A. Dhamdhere, and C. Dovrolis. "Open Issues in Router Buffer Sizing," *ACM SIGCOMM Comp. Communications Review*, 36(1), 2006.

[26] N. Egi, M. Dobrescu, J. Du, K. Argyraki, B. C. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, L. Mathy, and S. Ratnasamy, "Understanding the Packet Processing Capability of Multi-Core Servers," *Intel Technical Report*, 2009.

[27] J. Postel, "User Datagram Protocol", *Internet RFC768*, 1980.

[28] T. Wolf, and M. A. Franklin, "Locality Aware Predictive Scheduling of Network Processors", *Proc. of ISPASS*, 2001.

[29] J. E. Smith, and A. R. Pleszkun. "Implementation of Precise Interrupts in Pipelined Processors," *Proc. of ISCA*, 1985.

[30] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. "CACTI 5.1. Technical Report," *HPL-2008-20*, 2008.

[31] A. Bakhoda, G. L. Yuan, W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," *Proc. of ISPASS*, 2009.

[32] B. Bloom. "Space/Time Trade-offs in Hash Coding With Allowable Errors," *Communication of the ACM*, vol. 13, pp. 422-426, 1970.

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

[34] Tcpreplay, http://tcpreplay.synfin.net/trac.

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

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

[37] Y. Luo, L. Bhuyan, and X. Chen, "Shared Memory Multiprocessor Architectures for Softwre IP Routers," *IEEE Trans. Parallel and Distributed Systems*, vol.14, no. 12, 2003.

[38] MAWI Working Group Traffic Archive,

http://mawi.wide.ad.jp/mawi.

[39] H. J. Lee, M. S. Kim, W. K. Hong, and G. H. Lee, "QoS Parameters to Network Performance Metrics Mapping for SLA Monitoring," *KNOM Review*, vol. 5, no. 2, 2002.

[40] G. Armitage, Quality of Service in IP Networks, *Sams Publisher*, 2000.

[41] N. B. Lakshminarayana, and H. Kim, "Effect of Instruction Fetch and Memory Scheduling on GPU Performance," *Workshop on Language, Compiler, and Architecture Support for GPGPU, in conjunction with HPCA/PPoPP*, 2010.

[42] M. Dobrescu, N. Egi, J. Du, K. Argyraki, B. C. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "RouteBricks: Exploiting Parallelism To Scale Software Routers," *Proc. of SOSP*, 2009.

[43] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," *ACM Trans.Computer Systems*. 18(3), 2000.

[44] L. De Carli, Y. Pan, A. Kumar, C. Estan, and K. Sankaralingam. "PLUG: Flexible Lookup Modules for Rapid Deployment of New Protocols in High-speed Routers," *Proc. of SIGCOMM*, 2009.

[45] V. F. Pavlidis, and E. G. Friedman. "Three-Dimensional Integrated Circuit Design," *Morgan Kaufmann Publisher*. 2009.

[46] J. McCann, S. Deering, J. Mogul, "Path MTU Discovery," *Internet RFC 1191*, 1996.

[47] H. Wong, A. Bracy, E. Schuchman, T. M. Aamodt, J. D. Collins, P. H. Wang, G. Chinya, A. K. Groen, H. Jiang, and H. Wang, "Pangaea: A Tightly-Coupled IA32 Heterogeneous Chip Multiprocessor," *Proc. of PACT*, 2008.

[48] S. Melvin, and Y. Patt, "Handling of Packet Dependencies: A Critical Issue for Highly Parallel Network Processors," *Proc. of CASES*, 2002.

[49] S. Hong, and H. Kim, "An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness," *Proc. of ISCA*, 2009.

[50] S. Hong, and H. Kim, "An Integrated GPU Power and Performance Model," *Proc. of ISCA*, 2010.

Yuhao Zhu received the B.E. degree in computer science and engineering from Beihang University, Beijing, China. He is pursing the Ph.D. degree in electrical and computer engineering at the University of Texas at Austin. His current research interests include processor microarchitecture, computer system, and large-scale datacenter design.

Yubei Chen biography appears here. Degrees achieved followed by current employment are listed, plus any major academic achievements.

Yangdong Deng received his Ph.D. degree in Electrical and Computer Engineering from Carnegie Mellon University, Pittsburgh, PA, in 2006. He received his MS and BE degrees in Electronic Department from Tsinghua University, Beijing, in 1998 and 1995, respectively. He was with Incentia design Systems in 2004 as a senior software engineer (before finishing his Ph.D. dissertation). From 2006 to 2008, he was with Magma Design Automation as a consulting technical staff. Since 2008, he has been with Institute of Microelectronics, Tsinghua University, as an associate professor. He serves on the Technical Program Committees of many conferences. He is the founding co-chair of the First International Workshop on Frontier of GPU Computing and the co-chair of the second International Workshop on Frontier of GPU Computing. His research interests include electronics design automation, parallel programming, and GPU microarchitecture. His research is supported by the EDA Key Project of China Ministry of Science and Technology, Tsinghua-Intel Center of Advanced Mobile Computing Technology, China National Science Foundation, Intel University Programs, Tsinghua CUDA Center of Excellency, NVidia Professor Partnership Awards, and others. He is the author or co-author of 4 books and around 40 papers. He received an award from Magma Design Automation for his technical contributions in 2006. He is the chief invited expert of China National Excellent Class Center for GPU based parallel computing. He is a member of the IEEE and the IEEE Computer Society.