Optimization of Stateful Hardware Acceleration in Hybrid Architectures

Xiaotao Chang[1], Yike Ma[4], Hubertus Franke[2], Kun Wang[1], Rui Hou[1], Hao Yu[2], Terry Nelms[3]
[1]IBM Research-China, [2]IBM Watson Research Center, [3]IBM Software Group
[4]Institute of Computing Technology, Chinese Academy of Science
{changxt, wangkun, hourui}@cn.ibm.com, {frankeh, yuh, tnelms}@us.ibm.com, ykma@ict.ac.cn

***Abstract*—In many computing domains, hardware accelerators can improve throughput and lower power consumption, instead of executing functionally equivalent software on the general-purpose micro-processors cores. While hardware accelerators often are stateless, network processing exemplifies the need for stateful hardware acceleration. The packet oriented streaming nature of current networks enables data processing as soon as packets arrive rather than when the data of the whole network flow is available. Due to the concurrence of many flows, an accelerator must maintain and switch contexts between many states of the various accelerated streams embodied in the flows, which increases overhead associated with acceleration. We propose and evaluate dynamic reordering of requests of different accelerated streams in a hybrid on-chip/memory based request queue in order to reduce the associated overhead.**

I. INTRODUCTION

Network-optimized applications are constrained by ingress and egress data rates, latency and throughput requirements, and the temporal or streaming nature of the data as the current link rates are approaching 40Gb/s. To address the computational and I/O demands of this domain, processor chips are increasingly built as Systems on a Chip (SoC), integrating massively multi threading cores with network interfaces and generic and application specific accelerators onto a single chip [1].

Acceleration opportunities present themselves at various points in packet processing. The first opportunity is when a packet arrives. All packets that enter the system must be buffered, parsed, and scheduled on hardware threads for further processing. Accelerating these tasks can significantly increase performance. Decryption is also an opportunity for acceleration. Packets can be encrypted at any layer, but the most common locations are at the internet layer (IPSec) and at the application layer (SSL/TLS). Decryption algorithms can typically be implemented more efficiently in hardware than in software. Another acceleration opportunity is decompression. Compressed data are usually found at the internet layer and at the application layer. At the internet layer data are compressed before being encrypted (IPSec). This is commonly used with VPNs to reduce the required bandwidth for low-speed WAN links. At the application layer, HTTP 1.1 allows compressed content and is commonly used to reduce the bandwidth requirements for pages that are referenced often. In addition, large file transfers over email and the web are frequently compressed. Lastly, regular expression pattern matching (RegX) is commonly accelerated. Applications that use RegX regularly (e.g., intrusion detection/prevention, data leakage prevention) can experience a large performance boost when accelerated.

978-3-9810801-7-9/DATE11/©2011 EDAA

In many usage scenarios, acceleration typically works on one whole block of data at the application level after the data stream has been reassembled and different subsequent acceleration requests do not share any state. This kind of processing is referred to as stateless. Compared to stateless accelerators, stateful accelerators do not need to wait for all the packets of a flow to be reassembled. It can process any packet of the flow as it arrives. For instance intrusion detection systems emulate the application layers and there is a need to perform this emulation on a packet by packet basis. As a result, the high flow concurrency of network traffic extends to the accelerator module, requiring it to be stateful, i.e. the accelerator has to maintain a context for each accelerated stream associated with a flow when processing a packet of that flow. Stateful accelerator examples are all units that follow a stream model. These include compression and decompression, cryptography, RegX, and Extensible Markup Language (XML) processing.

Dependent on the accelerator, frequent context switching can introduce performance overhead, especially when the context size is large. In general, requests are fetched from a queue and processed by the accelerator in FIFO order. Our approach to reducing this overhead is to focus on the acceleration request queue and reordering requests to reduce the number of required context switches. In particular, our contributions are the reordering design of a hybrid on-chip/memory based request queue with limited resources, and the simulation based performance evaluation of this design.

The remainder of this paper is organized as follows: Section 2 describes the common design of a hardware request queue, based on which, Section 3 presents the design methods to optimize for stateful hardware acceleration. Section 4 presents the experimental setup, followed by simulation results.

II. BACKGROUND

When through the processing of the network stack sufficient data has accumulated to start an acceleration request (and this can be on a per packet base), the processing thread creates a coprocessor request block (CRB[1]) that in general includes the source data address, the target address and the context address. The context is a unique memory area, referred to as context ID or CID, that is allocated by the software associated with the specific accelerator and where the accelerator

maintains its state for an accelerated stream. The thread then dispatches the CRB to the accelerator to execute. The quasi concurrent submission from many threads can lead to the accelerator being busy and hence CRBs typically need to be queued up. In addition, the thread can asynchronously submit CRBs, which allows the thread to continue computation, potentially leading to further CRBs.

We expect that dependent on the architecture, a hardware request queue can potentially hold a thousand entries. However, due to limited on processor chip area, it is unfeasible to maintain the entire request queue on chip. Since millions of packets can arrive per second in current networks and due to well known bursty network behavior, it is easy to temporarily exhaust the on-chip queue capacity. Once the queue is full, threads can no longer submit additional CRBs and instead need to wait until previous CRBs completed. One solution is to utilize an additional spill queue, which is located in off-chip memory. Additional requests are spilled into this spill queue.

Once the spill queue technique is utilized for acceleration, the entire queue (on-chip plus off-chip) can be regarded as an infinite queue. According to queuing theory [2], this model is a standard $M/M/1$ model. We will use $M/M/1$ models as trace generators to conduct experiments in subsequent sections.

Due to the CRB interleaving of various streams, the network workload forces the accelerator to maintain the context of each stream for every CRB. In addition, due to the limited silicon area of the processor chip and the large size of the context for some accelerator (decompression needs 32KB of context), it is neither an option to keep the contexts of all active streams in hardware nor the entire context in some cases. Consequently, main memory needs to be utilized to maintain the context.

However, frequent stream context switching can introduce significant delays due to the memory wall. This presents an opportunity for optimization that we explore in the next section.

III. Request Reordering for Stateful Hardware Acceleration

To reduce the number of accelerator context switching and thus the overhead associated with the context save and restore memory operations, we propose to dynamically reorder the request queue in order to collocate pending requests of the same stream. To facilitate this, we describe a request reordering technique (ORR) for the on-chip queue and a spill request lookup (SRL) technique for the spill queue. One property that we must guarantee is that reordering maintains stability, i.e. that CRBs within the same stream remain in the same relative order.

The basic operation of ORR is described as follows: once a new incoming CRB arrives, it will be compared to all pending CRBs in the on-chip request queue based on the CID. If no match exists, the new CRB is inserted at the tail of the queue. If it matches any pending CRBs, it will be recorded in the on-chip-request reorder table (ORT) as the last CRB of that stream. The result is that when the accelerator finishes one CRB and further CRBs are pending, it will continue fetching the next CRB for the same stream utilizing the ORT, thus eliminating the need for a context switch.

When the on-chip queue is exhausted due to overload, the spill queue will be utilized to hold extra requests. Since the ORR can only detect the on-chip requests, the requests in the spill queue are always kept in submission order. However, once these requests are fetched from memory, they can be reordered based on ORT to reduce the overhead associated with acceleration. Unfortunately, if a large number of concurrent streams (CS) are in flight, there will be limited opportunities to reorder. For instance, if CS is larger than the on-chip queue size and the requests of each stream arrive to the queue in close to alternate order, the ORT will be of very limited benefit. We therefore propose another approach to reorder requests in the spill queue.

If we regard the on-chip queue as a window, all CRBs in the spill queue fall outside this window. Obviously, the larger the window size, the more opportunities are present for reordering. The ORR provides a limited opportunity to all pending on-chip CRBs to be reordered due to its limited window size. We therefore define an additional window that represents the information that covers the spill queue. We extend the design of the spill queue control block with CAM-based table, named stream lookup table (SLT).First, the spill queue is partitioned logically into many CRB groups, each of which has a fixed number of CRB slots. All CRBs in the same group belong to the same stream. Secondly, each entry in the SLT is comprised of one CID, one group address for the CRB group in the spill queue and one offset for the next empty request slot. Once a new CRB arrives, its CID is compared to the CID field of all valid SLT entries. If it matches one entry, this CRB will be spilled to the empty slot of that CRB group. The empty slot address can be obtained based on the group address and the slot offset in the matched SLT entry. If the new incoming CRB does not match any SLT entries, the SLT will allocate and populate a new entry. Meanwhile, this CRB is spilled to the first slot of that corresponding group.

Once one CRB finishes in the accelerator, the next CRB from the on-chip queue is fetched to process. In the case of a previously full on-chip queue, it leaves hence one empty entry for the CRB located in the first slot of the oldest CRB group of the spill queue to be fetched. Subsequently, all other CRB in this group will be fetched one by one every time an on-chip queue entry becomes available. When there is no more CRBs in the current group, it will be freed. The number of context switching can be reduced significantly since most successive CRBs have been already merged to the same group.

The SRL further extends the reordering window size. As the simulation results will demonstrate, the ORR can only provide benefits to smaller case, while the SRL can provide benefits in higher concurrency cases.

IV. Experimental Results

To demonstrate the efficiency of the proposed mechanism, we conduct a simulation based evaluation. We focus on a detailed performance study on the decompression accelerator

and only present high level results for the other accelerators. The performance study targets two aspects: throughput and response time.

For our analysis we first obtain the approximate service times that are based on acceleration and context switching cost. We assume an average payload of 1200 bytes per packet, an average compression ratio of factor 4.91 [5] and a context state of 2.5KB (2KB history [5] and 0.5KB dynamic Huffman table) that needs to be loaded when multiple CRBs per stream are present. There is no cost associated with swapping out a state because that is inherent in the cost of writing the history and output buffer of average 5892B (1200B*4.91) per CRB. These parameters were obtained through the Mambo Simulator for the IBM WSP [1][4], a wire-speed processor, combining 16 multithreaded IBM PowerPC cores with special-purpose dedicated accelerators. Mambo is an IBM full-system simulator. Both functional and cycle accurate simulation support for all four accelerators mentioned in WSP processor is integrated in Mambo.

The accelerator can decompress at approximately 1B/cycle when neither load nor write stalls are present [4]. Therefore, the cycles required for an acceleration request are estimated at the maximum of 1200 cycles for acceleration. Since 8 cache line (64B) stores takes 300 cycles in WSP, 3452 cycles will be taken for writing back 5892B of output. For a context switch in, we use 1464 cycles to load 2.5KB context. Following common queuing theory of $M/M/1$ with Poisson distribution, we create request arrival traces with varying numbers of concurrent streams ($CS = 8$) to ($CS = 2048$) and various submission rates to simulate increasing load on the various studied systems of FIFO and varying ORT sizes and SLT sizes. Each trace contains 100K CRBs.

A. Throughput Optimization

We first statically analyze the traces with different window sizes, from 1 entry (FIFO) to 128 entries, as shown in Figure 1, to determine the number of reordering that took place. The reordering count of default FIFO means the number of two successive CRBs, which belong to the same stream. As the CS increases, the reordering count will decrease in the same window size. At the same time, the reordering count increases as the window size increases. In some cases, the reordered CRBs can be up to 90% of all CRBs, which can lead to substantial performance optimization we analyze next.

Figure 2 shows the throughput optimization of the decompression accelerator with ORR only. With the reorder table size increasing, the throughput increases up to 26.7%. The worst case ($CS = 2048$) of throughput increasing is 2% when the CS is significantly larger than the window size.

We now evaluate the impact of SRL for the spill queue. We configure the slot number in one CRB group to 10 and consider SLT sizes of 128 and 512 entries. For the SLT scenarios, we fix the ORT to 128 entries from here on. Figure 3 shows the increased throughput obtained through the addition of SRL, especially as the CS increases. As expected, as long as CS is smaller equal than the underlying SLT size, this

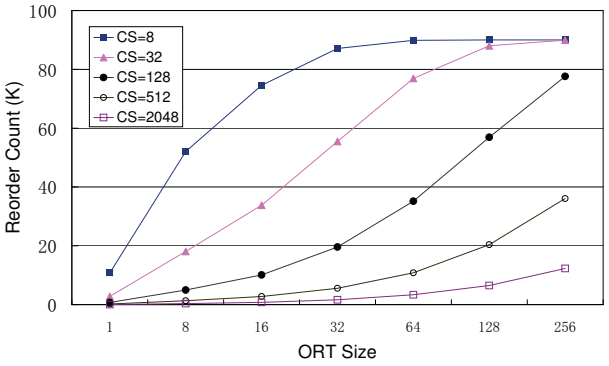


Fig. 1. Reorder Count with ORR

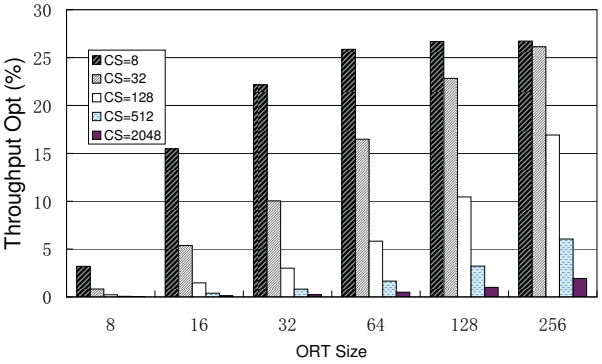


Fig. 2. Decompress Throughput Improvement with ORR over FIFO

mechanism is quite effective leading to over 20% performance improvements. Even as the concurrent streams are 4 fold that of the SLT table we still obtain a 7.2% performance improvements over FIFO.

Besides the decompression accelerator, we also conducted the throughput analysis for the other accelerators in the WSP processor based on their context size and based on their estimated service time considering acceleration and context switching overhead. Rather than presenting the detailed analysis, we limit the results in Table I, which shows the peak throughput gain respectively with utilizing the request reordering.

The throughput improvement of cryptography acceleration amounts to only 4%, due to its limited conctext size of 64 bytes. Consequently, its overhead for the context switching is not time consuming compared to the processing time, which limits its throughput improvement.

TABLE I
PEAK THROUGHPUT GAIN ON DIFFERENT ACCELERATORS

Accelerator	Context Size in IBM WSP (byte)	Throughput Improvement (%)
Decompression	2500	26.7
XML	256	15.8
RegX	192	9.9
Cryptography	64	4.0

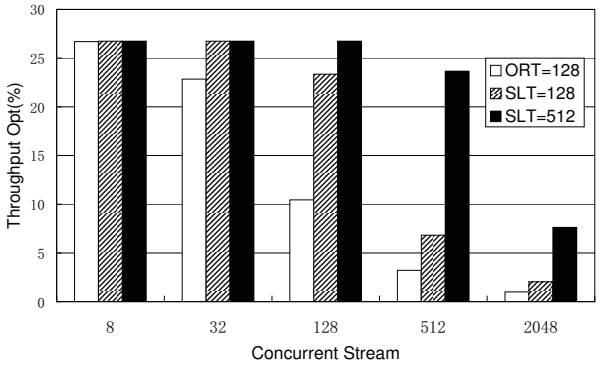
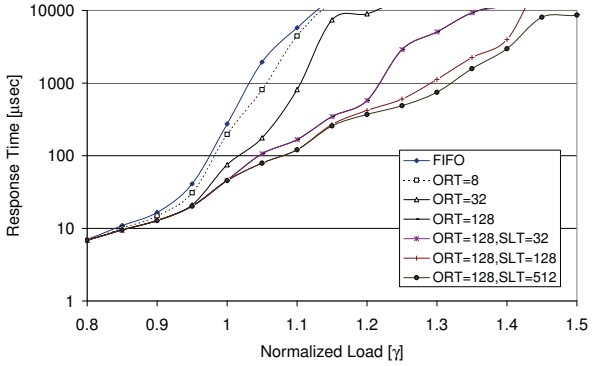


Fig. 3. Decompress Throughput Improvement with SRL over FIFO

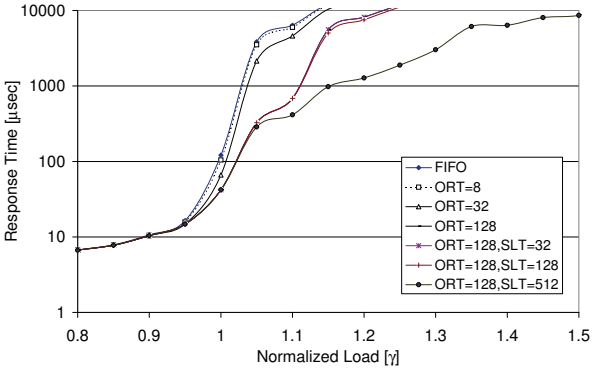
B. Response Time Optimization

In this section, we are evaluating the benefits of the request reordering with respect to the response time. In many applications, response time is critical in achieving the function of the application. As compared to the throughput analysis above, in response time sensitive applications (e.g. intrusion detection devices), we can not drive the accelerators to their maximum service rate. Using the trace generation method described earlier, we increase the request submission rate λ and measure the average response times. We report the response times as a function of the normalized load γ which is defined as $\gamma = \frac{\lambda}{\mu_{FIFO}}$, i.e. normalized to the load under the FIFO service rate.

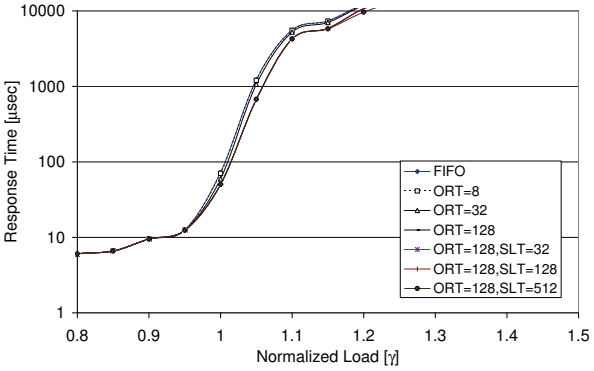
We investigate the following concurrent stream scenarios, namely 128, 512 and 2048 accelerated concurrent streams. Figures 8(a), 8(b), 8(c) show the impact of increased normalized load (γ) on the response time for those scenarios given different window size configurations of the ORT and SLT. Through these graphs we observe that in the low concurrent stream scenario ($CS = 128$) there is essentially no benefits to be observed until a normalized load of $\gamma > 0.8$ is reached, after which increasing ORT and SLT sizes provides benefits. For instance for $\gamma = 0.95$, $ORT = 128$ and all SLT configurations provide a 50% response time reduction. The addition of SLT provides additional benefits for $\gamma > 1.0$. At $\gamma = 1.2$, $SLT = 512$ provides an additional 36% reduction in response time over $ORT = 128$. For the medium concurrent stream scenario ($CS = 512$), ORT alone is mostly ineffective even though some response time reduction can be observed. With $\gamma > 1.05$ we still observe a 12% reduction for $SLT = 512$. Finally for the high concurrency case ($CS = 2048$) we see no benefits for load case $\gamma <= 0.95$. At $\gamma = 1.0$ we see a 31% reduction in response time for all ORTs and SLTs configurations, for $\gamma = 1.2$ we observe a 10% reduction by adding the SLT to ORT. The results demonstrate that if CS is higher than the ORT or SLT sizes, then our approach is of limited benefit.



(a) $CS = 128$



(b) $CS = 512$



(c) $CS = 2048$

Fig. 4. Impact of γ on Response Time

REFERENCES

[1] H. Franke, T. Nelms, H. Yu, H. D. Achilles, and R. Salz, B; Exploiting heterogeneous multicore-processor systems for high-performance network processing, IBM J. R&D, vol. 54, no. 1, Paper 2:1-14, 2010

[2] Robert B Cooper, Introduction to queueing theory (Second edition), North Holland; 1981

[3] Kannikar Siriwong, Lester Lipsky, Reda Ammar; Study of Bursty Internet Traffic, Sixth IEEE International Symposium on Network Computing and Applications (NCA 2007), 2007, pp.53-60,

[4] C. Johnson, D. H. Allen, J. Brown, S. Vanderwiel, R. Hoover, H. Achilles, C-Y. Cher, G. A. May, H. Franke, J. Xenedis, C. Basso; A Wire-Speed PowerTM Processor: 2.3GHz 45nm SOI with 16 Cores and 64 Threads; 2010 IEEE Int. Solid-State Circuits Conf; pp.104-105

[5] H. Yu, H. Franke, G. Biran, A. Golander, T. Nelms, and B. Bass; Stateful hardware decompression in networking environments, 4th ACM/IEEE Symp. Archit. Netw. Commun. Syst.,San Jose, CA, 2008, pp.141-150.