QWEB: High-Performance Event-Driven Web Architecture With QAT Acceleration

Jian Li, *Member, IEEE*, Xiaokang Hu, David Qian, Changzheng Wei, Gordon McFadden, Brian Will, Ping Yu, Weigang Li, and Haibing Guan, *Member, IEEE*

Abstract—Hardware accelerators have been a promising solution to reduce the cost of cloud datacenters. This article investigates the acceleration of an important datacenter workload: the web server (or proxy) that faces high computational consumption originated from SSL/TLS processing and HTTP compression. Our study reveals that for the widely-deployed event-driven web architecture, the straight offloading of SSL/TLS or compression tasks suffers from frequent blockings in the offload I/O, leading to the underutilization of both CPU and accelerator resources. To achieve efficient acceleration, we propose QWEB, a comprehensive offload solution based on Intel QuickAssist Technology (QAT). QWEB introduces an asynchronous offload mode for SSL/TLS processing and a pipelining offload mode for HTTP compression, both allowing concurrent offload tasks from a single application process/thread. With these two novel offload modes, the blocking penalty is amortized or even eliminated, and the utilization rate of the parallel computation engines inside the QAT accelerator is greatly increased. The evaluation shows that QWEB provides up to 9x handshake performance with TLS-RSA (2048-bit) over the software baseline. Additionally, the secure data transfer throughput is enhanced by 2x for the SSL/TLS offloading only, 3.5x for the compression offloading only and 5x for the combined offloading.

Index Terms—Accelerator, event-driven web architecture, SSL/TLS, HTTP compression, Offload I/O, concurrency

1 Introduction

Nowadays, cloud datacenters demand continuous performance enhancement to fulfill the requirements of cloud services that are becoming globalized and scaling rapidly [1], [2], [3]. Hardware accelerators, including GPU, FPGA and ASIC, are a promising solution to optimize the Total Cost of Ownership (TCO) as both energy consumption and computation cost can be reduced by offloading similar and recurring jobs [3], [4], [5], [6].

This paper targets an important type of datacenter workload: the web server (or proxy, e.g., reverse proxy). In general, today's web or proxy servers face two major sources of computational resource consumption: (i) the 'S' of HTTPS (i.e., SSL/TLS protocols [7]) that has been adopted by 60.9 percent of the top one million websites on the Internet [8]; (ii) the compression of HTTP content [9] that has been used by 81.8 percent of all the websites [10]. SSL/TLS processing involves continual crypto operations, particularly the expensive asymmetric ones, to ensure secure communication [11], [12], [13]. HTTP compression typically leverages the costly lossless algorithm with high compression ratio (e.g., gzip) [10] to save network bandwidth and improve end-user experience [14], [15].

• J. Li, X. Hu, and H. Guan are with Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: {li-jian, hxkcmp, hbguan}@sjtu.edu.cn.
• D. Qian, C. Wei, G. McFadden, B. Will, P. Yu, and W. Li are with Intel Corporation, Shanghai 200241, China. E-mail: {david.qian, changzheng.wei, gordon.mcfadden, brian.will, ping.yu, weigang.li}@intel.com.

*Manuscript received 24 Sept. 2019; revised 3 Mar. 2020; accepted 19 May 2020. Date of publication 2 June 2020; date of current version 16 June 2020. (Corresponding author: Jian Li.)
Recommended for acceptance by R. Ge.
Digital Object Identifier no. 10.1109/TPDS.2020.2999353*

A number of studies have made efforts to offload crypto or compression tasks to general-purpose accelerators, such as GPU [5], [16], [17], [18], [19], [20], [21], FPGA [22], [23], [24], [25], [26] and Intel Xeon Phi processor [13]. These studies mainly concentrated on the accelerator programming to enable efficient calculation of the needed crypto or compression algorithms, without paying much attention to the offload I/O. However, our study reveals an important fact: for the widely-deployed event-driven[1] web or proxy servers, such as Nginx [27], HAProxy [28] and Squid [29], the straight offloading of crypto or compression operations is not sufficiently competent for the web acceleration. The main challenge is the frequent blockings in the offload I/O, likely leading to (i) a large amount of CPU cycles spent waiting and (ii) a low utilization of the parallel computation units inside the accelerator. Since both CPU and accelerator resources are underutilized, the anticipated performance enhancement cannot be reached.

In this paper, we propose QWEB, a comprehensive offload solution based on Intel QuickAssist Technology (QAT) [30], to achieve high-performance SSL/TLS and compression acceleration for the event-driven web architecture. As a modern ASIC for both security (cryptography) and compression, the QAT accelerator has advantages on energy-efficiency and cost-performance compared to those general-purpose ones [1], [2]. Although the design of QWEB is based on Intel QAT, its main idea is applicable to other types of accelerators.

To address the blocking challenge in the offload I/O, QWEB is designed to allow *concurrent* offload tasks from a

1. handling multiple concurrent connections in one process/thread, instead of thread-per-connection, to achieve high performance

single application process/thread so as to (i) amortize or even eliminate the blocking penalty (i.e., CPU cycles spent waiting) and (ii) increase the utilization the parallel computation engines inside the QAT accelerator. Specifically, QWEB consists of an asynchronous offload mode for SSL/TLS processing and a pipelining offload mode for HTTP compression.

The SSL/TLS asynchronous offload mode re-engineers the SSL/TLS software stack to enable asynchronous support for crypto operations at all layers and completely eliminate the offload blockings. In this novel mode, CPU resources are fully utilized to handle concurrent connections and multiple crypto operations from different SSL/TLS connections can be offloaded concurrently in one application process. Moreover, this mode is built with (i) a heuristic polling scheme to retrieve QAT responses efficiently and timely and (ii) a kernel-bypass notification scheme to avoid expensive user/kernel mode switches while delivering events.

The compression pipelining offload mode takes advantage of the stream feature of HTTP compression to split the accumulated content into multiple pipelines for concurrent offloading. Although each offload request is still conducted in a synchronous/blocking way, the whole blocking penalty is amortized over the concurrent pipelines. Compared to a complicated asynchronous offload mode, this pipelining mode avoids heavy modifications into the web processing architecture and our evaluation demonstrates that it has a small performance gap from an asynchronous mode.

We have implemented the QWEB prototype based on the popular Nginx [27], which, including variants like Cloudflare Server [31], has been deployed by more than 50 percent of the top one million websites [32]. The two offload modes of QWEB were implemented as dynamic modules and can be customized directly from the Nginx configuration file. Particularly, we extended the simple SSL engine setting scheme in Nginx to an SSL Engine Framework so as to facilitate the integration of other accelerators. The implemented QWEB prototype has been available as open source [33] and adopted by a number of service providers, such as Alibaba for e-commence [34] and Wangsu for CDN [35].

The performance advantages of QWEB have been validated through extensive experiments that covered both the single offloading of SSL/TLS or compression and the combined offloading of these two. It's demonstrated that QWEB greatly improves the SSL/TLS handshake performance, achieving up to 9x connections per second (CPS) with the TLS-RSA (2048-bit) cipher suite over the software baseline. In addition, the secure data transfer throughput is enhanced by a factor of two for the SSL/TLS offloading only, a factor of 3.5 for the compression offloading only and a factor of five for the combined offloading.

In summary, this work makes the following contributions:

1) An important fact is revealed: for the widely-deployed event-driven web or proxy servers, the straight offloading suffers from the frequent blockings in the offload I/O.
2) We propose the high-performance SSL/TLS asynchronous offload mode to eliminate the blockings and allow concurrent offloading of crypto tasks, along with a heuristic polling scheme and a kernel-bypass notification.

TABLE 1
Server-Side Crypto Operations for Full Handshake

Protocol	Cipher Suite	RSA	ECC	PRF/HKDF
TLS 1.2	TLS-RSA	1	0	4
TLS 1.2	ECDHE-RSA	1	2	4
TLS 1.2	ECDHE-ECDSA	0	3	4
TLS 1.3	ECDHE-RSA	1	2	> 4

3) We design a high-performance pipelining offload mode for HTTP compression, which enables concurrent compression pipelines in a synchronous manner to effectively amortize the blocking penalty.
4) We show that QWEB can be practically implemented with Nginx and OpenSSL, and evaluate its performance with extensive experiments.

The remainder of this paper is organized as follows. We present background knowledge and highlight the challenges of hardware offloading in Section 2. The design of QWEB is elaborated in terms of SSL/TLS offloading and compression offloading in Section 3 and the prototype implementation based on Nginx is described in Section 4. Section 5 evaluates the performance advantages of the proposed QWEB. In Section 6, we review the related work, followed by a discussion in Section 7. Finally, we draw our conclusion in Section 8.

2 Background and Motivation

This section first analyzes the web-related computational cost. Then, we present background knowledge about the event-driven web architecture and Intel QAT. Finally, we highlight the challenges of hardware offloading for the event-driven web architecture.

2.1 Computational Cost of Web Servers

In general, today's web or proxy servers face two major sources of computational cost: the 'S' of HTTPS (i.e., SSL/TLS protocols) and the compression of HTTP content.

2.1.1 SSL/TLS

The 'S' of HTTPS relies on the SSL/TLS protocols, which have been de-facto standards for the Internet security for more than 20 years [36]. As reported in March 2020, 60.9 percent of the top one million websites on the Internet have enabled SSL/TLS based secure communication [8].

An SSL/TLS connection can be divided into two main phases [7]: (1) the handshake phase that determines crypto algorithms, performs authentications and negotiates shared keys for data encryption and message authentication code (MAC); (2) the secure data transfer phase that sends encrypted data over the established connection. In the handshake phase, multiple crypto operations are required at the server side, as summarized in Table 1. For the classic RSA-wrapped handshake (i.e., TLS-RSA), one asymmetric RSA signature is involved for authentication, which is the most computing-intensive part of the SSL/TLS processing [5], [11]. Besides, four pseudo random function (PRF) [37] operations are needed for key derivation. For other cipher suites with forward secrecy [38], such as ECDHE-RSA [39], the handshake is more complicated and involves two more asymmetric elliptic-curve cryptography (ECC) [40] calculations at the sever side. In the recently-released TLS 1.3 protocol [41], although one network round trip can be saved for handshake, the involved crypto operations cannot be omitted [42]. Also, the enhanced security requires more key derivation operations with the new HMAC-based key derivation function (HKDF) [43]. After a successful handshake, the server securely transfers data based on the negotiated cryptographic keys, including at least two cipher keys and two MAC keys. Each key pair is used for data encryption and MAC calculation in one direction. Prior to encryption and transmission, the data object is fragmented into units of 16 KB if it is larger than this.

In summary, SSL/TLS imposes a much higher resource consumption over an insecure implementation due to the costly crypto operations [5], [11], [12], [13]. A recent study [44] reported that a modern CPU core, which is able to serve over 30k HTTP connections per second, cannot handle more than 0.5k SSL/TLS handshakes per second with the mainstream ECDHE-RSA (2048-bit) cipher suite. The overhead mainly originates from the heavy asymmetric-key calculations [5], [11]. As the requested web object increases in size, the computation overhead incurred by data encryption and MAC calculation becomes considerable as well. Session resumption [7] is a technique that can be used to avoid performing the full handshake each time and reduce resource consumption, but it may compromise the protection afforded by forward secrecy [45]. In real-life deployment, service providers restrict the lifetime of session IDs or tickets (generally less than an hour [45]) to achieve a trade-off between performance and security.

2.1.2 HTTP Compression

HTTP compression [9] is a capability that can be built into web servers to allow HTTP content to be compressed before transmission to the client. It is considered as an essential tool in todays web [15], [46] and used by more than 80 percent websites on the Internet [10] because of the following benefits.

The compression of HTTP content largely reduces the amount of data sent over the network with a typical saving of 60-85 percent in bandwidth cost [15]. From the perspective of an end user, an improved experience can be obtained due to the reduced download time and page-load latency, especially in a poor network environment [14], [46]. When HTTP compression is enabled for SSL/TLS connections, it also deceases the amount of content that needs to be encrypted on the server and decrypted by the client.

However, these benefits from HTTP compression come at the expense of computational resource consumption. The dominant compression schemes in today's websites are gzip and deflate, which are used by nearly all the websites that enable compression [10]. Deflate [47] is a lossless data compression algorithm that combines LZ77 algorithm and Huffman encoding. Gzip [48] is actually a file format that introduces a header and a checksum on the basis of the deflate compression. The gzip/deflate compression can provide a high compression ratio, but it implies the consumption of a large amount of CPU cycles [49], [50]. Although static web pages are suitable for compress-once, distribute-many situations to decrease compression cost, the dynamic nature of Web 2.0 requires to compress pages on-the-fly for each client request [15]. There is no doubt that the heavyweight gzip/deflate compression puts a considerable burden on today's web servers [15].

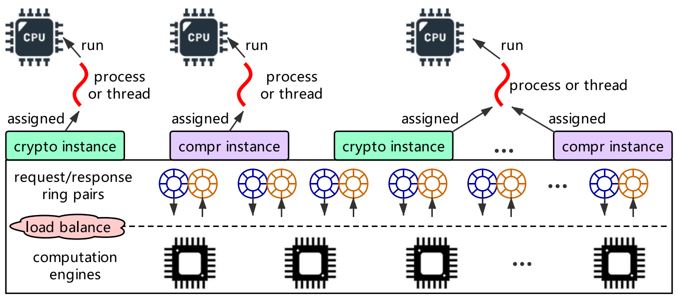


Fig. 1. A QAT endpoint and its usage model.

2.2 Event-Driven Web Architecture

There are two competitive architectures for web applications: one is based on threads, while the other is based on events [51]. Thread-based web applications (e.g., Apache [52]) initiate a new thread for each incoming connection. This suffers from costly thread switches and high system resource consumption under high traffic. In contrast, event-driven web applications (e.g., Nginx [27]) have the ability to handle thousands of concurrent connections in one process/thread, bringing significant advances in both performance and scalability [53], [54]. And to align with the multicore architecture, multiple worker processes can be launched to handle incoming connections in a balanced manner. Nowadays, the use of event-driven web architecture (e.g., Nginx, Cloudflare Server and LiteSpeed) has been nearly 55 percent among the top one million websites [32].

To consolidate multiple connections to a single flow of execution, the event-driven web architecture works with network sockets in an asynchronous (non-blocking) mode and monitors them using an event-based I/O multiplexing mechanism (e.g., epoll [55] or kqueue [56]). In the so-called event loop, the application process waits for events on the monitored sockets. Once a queue of new events is returned, the corresponding event handlers are invoked one by one.

2.3 Intel QuickAssist Technology

Intel QuickAssist Technology is an ASIC-based solution to enhance security and compression performance for cloud, networking, big data and storage applications [30]. Both kernel driver and userspace driver (implemented as a library) are available to meet different requirements.

Usage Model. As illustrated in Fig. 1, a QAT endpoint possesses a number of *parallel* computation engines and hardware-assisted *request/response* ring pairs. Software writes requests onto a request ring and reads responses back from a response ring. The availability of responses can be indicated using either interrupt or polling. In order to achieve a desired level of performance, implementations of a QAT endpoint include multiple computation engines for compression, symmetric cryptography and public key cryptography, respectively, all of which can operate in parallel [57]. The load balancing of requests from all rings across these

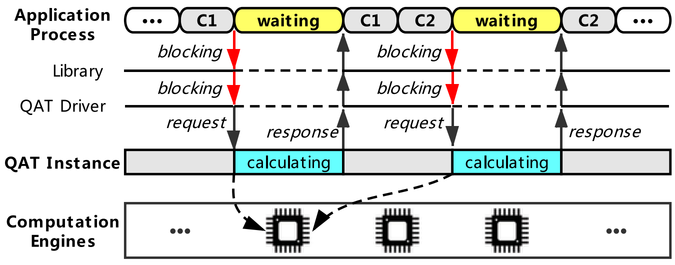


Fig. 2. Straight offload mode with QAT for event-driven web servers.

computation engines is performed by QAT hardware automatically.

Several request/response ring pairs are grouped into a QAT crypto or compression instance, which is actually a logical unit that can be allocated and assigned to a process/thread running on the CPU core. A modern QAT endpoint can support tens of crypto and/or compression instances (at most 128 instances) to scale with the multi-core architecture [58]. If there are multiple QAT endpoints on the machine, one process can be assigned with multiple QAT instances from different endpoints to employ more computation engines.

Parallelism. First, crypto or compression requests submitted from different processes to different QAT instances can be calculated in parallel on multiple computation engines. Second, *concurrent* crypto or compression requests submitted from one process to one QAT instance can also be calculated in parallel as long as there are available computation engines. Here, "concurrent" means to submit the next request without waiting for the completion of previous ones. If there are sufficient concurrent requests, it is possible to fully load all parallel computation engines with only a small number of QAT instances (e.g., 2 or 3).

OpenSSL QAT Engine. QAT integrates its crypto service into OpenSSL (the de facto standard for crypto library [59]) through a standard engine named QAT Engine [60], which allows the seamless use of QAT by applications. When the QAT Engine is loaded, applications can transparently offload crypto operations to the QAT accelerator via the normal OpenSSL API.

2.4 Challenges of Hardware Offloading

A straight offload mode for web servers is to replace the crypto and compression function call with an I/O call that interacts with the hardware accelerator. However, for the event-driven web architecture, this kind of straight offloading is not sufficiently competent and cannot reach the anticipated performance enhancement.

In the software-based event-driven web servers, the crypto or compression function call is a synchronous blocking call to use the CPU resource for calculation. It works well on the CPU-only architecture, but with the straight offload mode, it inevitably causes frequent blockings in the offload I/O, as illustrated in Fig. 2. The QAT driver provides an inherently non-blocking interface through request/response based communication. However, the I/O call for submitting an offload request cannot directly return control to upper layers as both the application and library cannot move on with an uncompleted crypto or compression job. In other words, the application process cannot continue execution until the QAT accelerator completes this request and generates a response.

For the event-driven web architecture, the frequent blockings in the offload I/O lead to the underutilization of both CPU and QAT resources. First, a large amount of CPU cycles are spent waiting. After a crypto or compression request is submitted to the accelerator, the application process (typically running on a dedicated core) is in the waiting state (either busy-looping or sleeping to wait for the QAT response), stopping the cycle of handling events for a long time. Second, the utilization rate of the parallel computation engines inside the QAT accelerator is low. Since the second offload request can only be submitted after the completion of the first one, for each application process, no more than one computation engine can be employed at the same time.

The blocking time for each offload job depends on the type of the offloaded task. Table 2 summarizes the QAT offload latency for different kinds of tasks involved in web servers. We measured the latency (i.e., blocking time) by continuously executing 2,000 synchronous offload operations in a tight loop directly from the QAT driver level. It can be observed that for the asymmetric RSA signatures, the average offload latency (i.e., blocking time) is up to 745 μs for the mainstream 2048-bit RSA key and 5332 μs for the emerging 4096-bit RSA key. With this long blocking time, there is still much room for performance improvement.

3 QWEB Design

To achieve high-performance SSL/TLS and compression acceleration for the event-driven web architecture, we propose QWEB, a comprehensive offload solution based on Intel QAT. Its design is also applicable to other types of accelerators.

3.1 Overview

The main idea of QWEB is to allow *concurrent* offload tasks from a single application process/thread so as to (1) amortize or even eliminate the blocking penalty (i.e., CPU cycles spent waiting) and (2) increase the utilization of the parallel computation engines inside the QAT accelerator. As shown

TABLE 2
QAT Offload Latency for RSA Signature, AES-128 Symmetric Chained Cipher and Deflate Compression

	RSA Signature			AES-128 Symmetric					Deflate Compression				
Latency	*RSA-1024*	*RSA-2048*	*RSA-4096*	*1KB*	*2KB*	*4KB*	*8KB*	*16KB*	*4KB*	*8KB*	*16KB*	*32KB*	*64KB*
Min. (μs)	153	745	5,331	8	10	12	17	26	27	39	64	108	193
Ave. (μs)	154	745	5,332	8	10	13	17	27	28	41	66	110	194
Max. (μs)	154	746	5,336	9	11	13	18	34	29	42	67	113	196

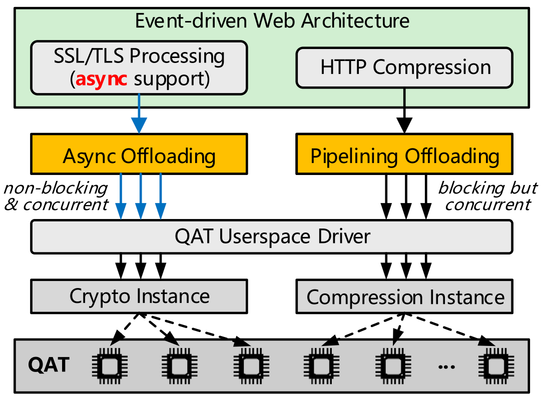


Fig. 3. Overall architecture of QWEB.

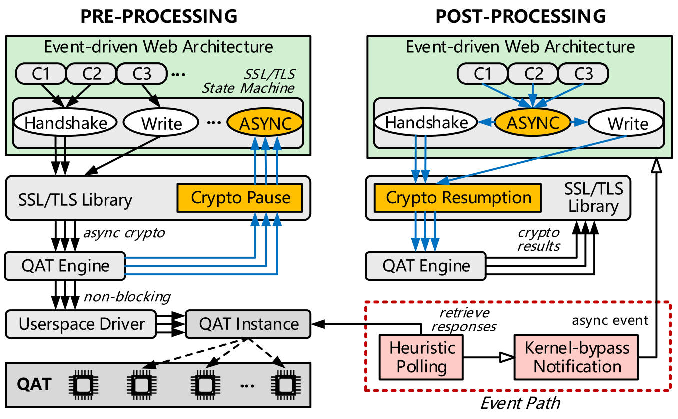


Fig. 4. Asynchronous offloading of SSL/TLS processing.

in Fig. 3, the QWEB architecture consists of an asynchronous offload mode for SSL/TLS processing and a pipelining offload mode for HTTP compression.

The SSL/TLS asynchronous offload mode re-engineers the SSL/TLS software stack to enable asynchronous support for crypto operations at *all layers*. It completely eliminates the offload blocking (i.e., introducing a non-blocking path) and makes multiple crypto operations from different SSL/TLS connections able to be offloaded concurrently in one application process. The details of this mode, along with the optimized event path consisting of a heuristic polling scheme and a kernel-bypass notification scheme, will be presented in Section 3.2.

The compression pipelining offload mode takes advantage of the stream feature of HTTP compression to split the accumulated content into multiple pipelines for concurrent offloading. Although each offload request is still conducted in a blocking way, the whole blocking penalty is amortized over the concurrent pipelines. The details of this mode and the reason why an asynchronous offload mode is not designed for compression will be elaborated in Section 3.3.

3.2 SSL/TLS Asynchronous Offload Mode

As shown in Table 2, the offload blocking time of one asymmetric crypto operation is up to 745μs for the mainstream RSA-2048 and 5332μs for the emerging RSA-4096. Due to this long blocking time, it is important and necessary to design a non-blocking path for SSL/TLS offloading.

3.2.1 *Pre-Processing and Post-Processing*

The root cause of the crypto offload blocking is that both the SSL/TLS library and the application cannot move on if a crypto operation has not been completed yet. We overcome this restriction by enabling asynchronous crypto support into the whole SSL/TLS software stack and dividing the offload I/O into a pre-processing phase and a post-processing phase, as illustrated in Fig. 4.

Pre-Processing. When an SSL/TLS handler (e.g., the handshake handler) encounters a crypto operation and offloads it, the invoked QAT Engine API first submits the crypto request to QAT accelerator and then *pauses* the offload job immediately to return the control to the application. The application process continues execution to handle other connections, instead of being blocked and waiting for the QAT response. In this way, multiple crypto operations from different SSL/TLS connections (C1, C2, ...) can be offloaded *concurrently* in one application process.

Post-Processing. As more and more concurrent crypto requests are submitted, the QAT responses for them (indicating ready results) need to be retrieved in time. QWEB leverages userspace based *polling* to retrieve QAT responses. Each time a new QAT response is retrieved, an *async event* is generated to inform the application about the completion of an async crypto request. According to the captured async event, the corresponding connection (e.g., C1) along with the same SSL/TLS handler (e.g., the handshake handler) will be rescheduled by the application to *resume* the paused offload job. Since the crypto calculation result is now ready, this connection can move on to the next step.

Requirements. This mode requires asynchronous support for crypto operations at *all layers* of the SSL/TLS software stack, including: the application layer, the SSL/TLS library layer and the QAT Engine layer.

In the SSL/TLS library layer, QWEB introduces *crypto pause* and *crypto resumption*. The former is leveraged by the QAT Engine to pause the current offload job after a crypto request is submitted and return control to the application with a specific error code. The latter is leveraged by the application process to resume a paused offload job to consume the crypto calculation result. The pause and resumption of an offload job require careful management of the SSL/TLS context and the crypto state.

The application layer is both the initiator and terminator of async crypto operations. QWEB introduces a new state named ASYNC into the application's SSL/TLS state machine. This state is connected to all other SSL/TLS states (e.g., Handshake and Write) that may involve crypto operations. Suppose that a connection C1 is in Handshake state (i.e., executing the handshake handler) during the pre-processing phase, as illustrated in Fig. 4. Once the SSL/TLS state machine identifies a paused offload job from the specific error code returned by the SSL/TLS library, it changes the connection's state from Handshake to ASYNC and equips the ASYNC state with an *async handler*. Here, the async handler is set to the handshake handler, indicating that the same handler needs to be rescheduled later to conduct the post-processing phase.

The QAT Engine layer is a bridge between the SSL/TLS library layer and the QAT driver layer. First, it is responsible to submit crypto requests to the QAT accelerator using the non-blocking API provided by the QAT driver. Second, after the submission of a crypto request, it uses the *crypto pause* interface provided by the SSL/TLS library to return control to the upper application.

3.2.2 Heuristic Polling

QWEB leverages polling, instead of interrupt, to retrieve QAT responses. The reason is that QWEB is based on user-space I/O (i.e., QAT userspace driver), where one userspace based polling operation has much less overhead than one kernel based interrupt [61]. The QAT Engine has provided a convenient polling method: initiate an independent thread for each application process to poll the assigned QAT instances at regular intervals. However, it comes with several drawbacks inevitably:

- It introduces frequent context switches between application processes and polling threads.
- It is hard to determine an appropriate polling interval: a small interval may lead to a large amount of ineffective polling operations while a big interval may incur a long latency or even impede the throughput.

QWEB adopts a *heuristic polling* scheme to achieve better performance and adapt well to varied traffic. Since the application is the producer of crypto operations, it has the best knowledge about the proper time to retrieve QAT responses. The heuristic polling scheme is integrated into the application to (1) avoid an independent polling thread and (2) leverage the application-level knowledge to guide the polling behaviors. It considers both efficiency and timeliness, with the goal to make the response retrieve rate always match the request submission rate.

On the one hand, when there are a large number of concurrent SSL/TLS connections (i.e., high offload traffic), it is reasonable to coalesce sufficient QAT responses into one polling operation. The number of *inflight* (submitted but not retrieved with a response) crypto requests is an appropriate indicator and a polling operation is triggered when this number reaches a threshold. Considering the fact that the asymmetric crypto request takes a much longer calculation time than other types of crypto requests, a bigger threshold could be used if there exist inflight asymmetric crypto requests. On the other hand, during off-hours with few SSL/TLS connections, a timely polling is necessary. An SSL/TLS connection can be identified as active if it is processing the handshake, reading a request or writing a response back. Since each active SSL/TLS connection can only have one async crypto request at the same time, once the number of inflight crypto requests equals the number of active connections, a polling operation needs to be executed immediately. Otherwise, the application process may get stuck as all active connections are waiting for QAT responses. This timeliness constraint greatly reduces the query latency when there are only few end clients.

3.2.3 Kernel-Bypass Notification

For the event-driven web architecture, a natural way for async event notification is to reuse the existing event-driven approach. Specifically, each time an async offload job is paused, a file descriptor (FD) [62] is allocated to it and monitored by the application's I/O multiplexing mechanism, together with the network sockets. When a QAT response is retrieved, an event is written on the corresponding FD to notify the application. However, since FDs are maintained by kernel, this kind of notification inevitably introduces expensive switches between user mode and kernel mode.

QWEB adopts a kernel-bypass notification scheme to avoid this performance penalty. An application-defined *async queue* is introduced to work as the monitoring and notification channel. When the SSL/TLS state machine identifies a paused offload job, it already has the knowledge about its *async hander*, which is actually the SSL/TLS handler needed to be rescheduled during the post-processing phase. This knowledge can be shared to the lower layers of the software stack. Then, when a QAT response is retrieved, async event notification can be conducted by just inserting the corresponding *async handler* into the tail of the *async queue*. This *async queue* will be processed by the application at the end of the main event loop. As long as there exist inflight crypto requests, the main event loop keeps execution, instead of sleeping and waiting for events.

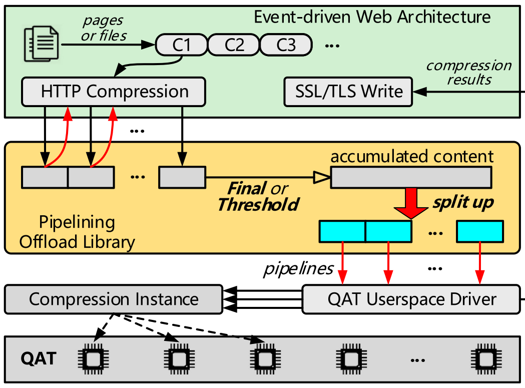


Fig. 5. Pipelining offloading of HTTP compression.

3.3 Compression Pipelining Offload Mode

QWEB introduces a pipelining offload mode, instead of an asynchronous one, for HTTP compression due to the following reasons:

- The stream feature of HTTP compression provides the opportunity to accumulate HTTP content and amortize the blocking penalty over multiple concurrent offload jobs.
- As shown in Table 2, the offload blocking time of a typical compression operation is between $20\mu s$ and $200\mu s$, not so high as that of an asymmetric crypto operation.
- Our evaluation demonstrates that this pipelining offload mode only has a small performance gap from an asynchronous one.
- An asynchronous offload mode for compression may incur heavy modifications (i.e., enabling asynchronous support) into the web processing architecture, as what the SSL/TLS asynchronous mode does.

The architecture of the compression pipelining offload mode is illustrated in Fig. 5. A pipelining offload library is introduced to replace the traditional software-based compression library (e.g., zlib [63]). The straight offload mode sends one chunk of data as one request to QAT and then waits for its completion, which means that, in one blocking time, only one request can be calculated using one computation engine in QAT. The core idea of this pipelining offload mode is to continuously send multiple chunks of data as multiple requests to QAT and then waits for their completion. These concurrent requests can be calculated in parallel inside QAT using multiple computation engines. From the perspective of hardware acceleration, these concurrent offload jobs greatly increase the utilization of the underlying QAT resources. From the perspective of application, since multiple chunks of data can be compressed in one (slightly longer) blocking time, the blocking penalty is amortized by them.

Stream Feature. The HTTP compression in web servers has a stream feature. When a connection obtains a part of content of an HTTP response, it immediately submits the content to the HTTP compression module (or more exactly, the compression library). If the compression library outputs the compression result at this invoking, the returned result will be further delivered for encryption or transmission. If there is no output result at this invoking, this connection continues to obtain the next part of content of the HTTP response and go on. The compression library outputs compression results at it own pace for better compression effect and flushes all results if the caller specifies that the currently submitted content is the *Final* one of the HTTP response.

Accumulate and Offload. The pipelining offload library first takes advantage of this stream feature to accumulate HTTP content. When the web application submits a part of content (e.g., 32 KB) of an HTTP response to the pipelining offload library, the library caches it without compression and then directly gives the control back to the application. The application continues execution and submits the next part of content of the HTTP response. The pipelining offload library performs compression only if (i) the currently submitted content is the *Final* one or (ii) the accumulated HTTP content has reached a pre-defined threshold (e.g., 256 KB). In these two cases, the accumulated HTTP content is split into multiple pipelines (e.g., 64 KB each) for concurrent compression offloading. Specially, the pipelining library submits each pipeline to QAT one after one continuously and then waits for the completion of all the submitted requests. Although each offload request is still conducted in a blocking way, the concurrently submitted requests could be calculated in parallel inside the QAT accelerator. In other words, the blocking penalty is amortized over the concurrent pipelines.

Our evaluation proves that the pipelining offload mode for HTTP compression provides an obviously higher performance over the straight offload mode. And the average CPU utilization of the pipelining mode is already above 80 percent. It means that even if we make efforts to enable an asynchronous offload mode that can fully utilize the CPU resources (i.e., 100 percent utilization), the extra performance advantage compared to the proposed pipelining mode is limited. Moreover, the incurred overhead of context switches among asynchronous offload jobs may further offset the advantage.

4 Implementation Details

QWEB can be implemented on various event-driven web applications (e.g., Nginx, HAProxy and Squid). This section presents the implementation based on the popular Nginx [27], which supports a number of working modes, such as a web server, a reverse proxy server or a load balancer. Nginx and its variants (e.g., Cloudflare Server [31]) have been used by more than 50 percent of the top one million websites [32]. The QWEB implementation based on Nginx has been available as open source [33].

As a typical event-driven web application, Nginx uses one master process for management and multiple independent worker processes for doing the real work. In Linux environment, each worker process leverages the epoll mechanism [55] to monitor and handle thousands of connections. QWEB is implemented into each worker process independently to perform crypto and compression offloading.

4.1 SSL/TLS Asynchronous Offloading

4.1.1 Async Crypto Behaviors

As presented in the design part, the SSL/TLS asynchronous offload mode requires asynchronous crypto support at *all layers* of the SSL/TLS software stack, including: the application layer (here is Nginx), the SSL/TLS library layer (here is OpenSSL[59]) and the QAT Engine layer.

OpenSSL Async Crypto Infrastructure. OpenSSL [59] is the de facto standard for SSL/TLS and crypto library. We have developed the *crypto pause* and *crypto resumption* in OpenSSL for several years. There are two kinds of implementations: stack async and fiber async. The stack async implementation achieves pure async behaviors by altering the normal sequence of program execution according to state flags (refer to our prior conference version [64] for details). It has a good performance but it is intrusive to OpenSSL as its API is not compatible with the existing synchronous API. The fiber async implementation unifies the asynchronous and synchronous API using the fiber mechanism [65] and has been included into OpenSSL official releases since version 1.1.0 series.

Fibers are cooperative lightweight threads of execution in user space, which, in effect, provide a natural way to organize concurrent code based on asynchronous I/O [66]. By using explicit yielding and cooperative scheduling, fiber async implementation manages multiple execution paths for async offload jobs in one process/thread, as illustrated in Fig. 6. When the SSL/TLS API (e.g., handshake) is called for the first time, it uses *ASYNC_start_job* to start a new fiber-based *ASYNC_JOB* to encapsulate (fiber context swap here) the running piece of the SSL/TLS connection. The fiber-based *ASYNC_JOB* can be paused at any point to return control to the main code and resumed later to directly jump to the pause point. If the invoked crypto API in the QAT Engine identifies an async offload job through *ASYNC_get_current_job*, it first submits a crypto request in a non-blocking mode and then leverages *ASYNC_pause_job* to return control (fiber context swap here). When the same SSL/TLS API is called again, it uses *ASYNC_start_job* with the paused ASYNC_JOB as an argument to directly jump (fiber context swap here) to the pause point (i.e., into the QAT Engine) and consume the crypto result.

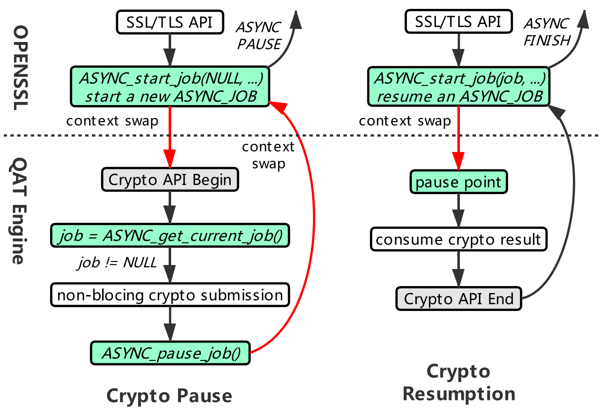


Fig. 6. Workflow of fiber async implementation.

Nginx Async Crypto Support. The crypto pause may occur in the following functions: *ngx_ssl_handshake*, *ngx_ssl_handle_recv*, *ngx_ssl_write* and *ngx_ssl_shutdown*. These functions are modified to recognize the new error code from OpenSSL named *SSL_ERROR_WANT_ASYNC*, which indicates that an async crypto request has been submitted. If this new error code is received, the async handler of the paused offload job is is set to the current handler (e.g., *ngx_ssl_handshake_handler*). Ideally, after an async offload job is paused and the execution is returned to Nginx, the only type of event expected for this SSL/TLS connection is an async event, which will resume the paused job. However, the possibility exists that a read event (from the network socket) occurs before the expected async event. This read event could be the next message during the SSL/TLS handshake or the first HTTP request after connection establishment. This kind of event disorder may cause the HTTP or SSL/TLS state machine to go back to a previous state. To address this issue, QWEB clears and saves the handler of the read event when an async event is being expected, and restores it when the async event is being processed.

4.1.2 Heuristic Polling

The information about the inflight crypto requests is collected in the QAT Engine layer for accuracy. The different types of crypto operations are counted independently, denoted by R_{asym}, R_{cipher} and R_{prf}, respectively. These information is shared to the upper application with a new engine command. Nginx has a *stub_status* module that collects useful information including the number of alive connections and the number of idle connections that are waiting for a request. Based on this, QWEB calculates the number of active SSL/TLS connections with the the equation: $C_{active} = C_{alive} - C_{idle}$.

The efficiency and timeliness constraints can be satisfied either when the number of inflight crypto requests increases or when C_{active} decreases. Therefore, in Nginx, wherever a crypto operation may be involved or C_{active} may be updated , it is required to check these constraints and determine whether to execute a polling operation. The default thresholds for the efficiency constraint are set to 48 (for $R_{asym} > 0$) and 24 (for $R_{asym} = 0$) based on our empirical evaluation, and we have opened the threshold setting in the Nginx configuration file. In our large amount of tests, the heuristic polling scheme works well to retrieve all QAT responses in time. For a failover, a timer (e.g., 5 μs) could be set inside Nginx to periodically check whether at least one heuristic polling operation is triggered during the last interval. If not but there exist inflight crypto requests, an extra polling operation can be executed at once.

4.1.3 Kernel-Bypass Notification

An application-level callback function is introduced to manipulate the *async queue*. This function takes the *async handler* information as the argument and inserts it into the tail of the async queue. In OpenSSL, two new members, named *callback* and *callback_arg*, are added to the data structure of the fiber-based *ASYNC_JOB*. Meanwhile, a series of new APIs are introduced for users to set or get them.

Each time the SSL/TLS state machine identifies a paused offload job, it invokes the *SSL_set_async_callback* API to set the application-level callback and the async handler. When a QAT response is retrieved, the response callback function leverages the *ASYNC_WAIT_CTX_get_callback* API to check whether the *callback* member has been set for the current offload job. If so, it can directly invoke the application-level callback function with *callback_arg* (i.e., the *async-handler* information) as the argument to complete notification.

4.2 Compression Pipelining Offloading

Nginx leverages the gzip format (i.e., the deflate algorithm) [48] for the compression of HTTP content. To take advantage of the stream feature of the HTTP compression in Nginx, the pipelining offload library introduces the concept of stream and session. That's to say, each HTTP response for a client request is regarded as a stream and equipped with a session for QAT offloading.

Before the first part of content of an HTTP response is submitted to the pipelining offload library, Nginx invokes the *stream_init* function to initialize this HTTP response stream for QAT compression. This function is responsible to (1) obtain and initialize QAT compression instances and (2) establish a QAT compression session for this stream, which determines the offload behaviors according to the user's configuration (e.g., compression level and the threshold for hardware/software switch). An optimization here is that for each Nginx worker process, only the generating of the first HTTP response triggers the allocation of QAT compression instances. Since the allocation and binding of QAT instances are at the level of process, the subsequent HTTP responses in the same worker process can reuse the obtained QAT instances for compression offloading.

After the above initialization work, Nginx invokes the synchronous *qzCompressStream* API to submit parts of content of the HTTP response. The pipelining offload library first stores the submitted content internally and then checks whether the size of the accumulated HTTP content reaches the pre-defined threshold (denoted as *qatzip_stream_size*). If not, the *qzCompressStream* API directly returns and Nginx continues execution to submit more HTTP content. If the *qatzip_stream_size* is reached, the accumulated HTTP content is split up into multiple chunks according to the size denoted by *qatzip_chunk_size*, and these chunks are submitted to the QAT accelerator one after one continuously. Afterwards, the pipelining offload library uses periodical polling (e.g., 5 or 10μs, which has tolerant influence on latency according to Table 2) to retrieve QAT responses. Once all QAT responses for the concurrently submitted requests are retrieved, the *qzCompressStream* API returns with ready compression results and Nginx further delivers the compressed HTTP content for encryption or transmission. Another case of triggering the offload action is that Nginx explicitly sets the *Final* flag in the *qzCompressStream* API to indicate that the submitted HTTP content is the last part of the current HTTP response. In this case, the pipelining offload library immediately splits the accumulated HTTP content for concurrent offloading, regardless of its size.

The performance of the compression pipelining offloading depends on the selection of *qatzip_stream_size* and *qatzip_chunk_size*, especially the ratio between them (i.e., the number of pipelines). A small *qatzip_chunk_size*, i.e., more pipelines, can increase concurrency but it incurs a higher offload overhead (e.g., preparing/consuming more requests/responses). A big *qatzip_chunk_size*, i.e., less pipelines, reduces the offload overhead but it likely suffers from a high blocking penalty and impedes the throughput. We used empirical evaluation based on the default Nginx configuration to select proper values for them, as presented in the next section. Finally, the default values for *qatzip_stream_size* and *qatzip_chunk_size* are set to 256 KB and 64 KB, respectively, to achieve a good offload performance. Similarly, we have enabled the setting of these two parameters from the Nginx configuration file.

4.3 Nginx Modules and Configuration

The modular architecture of Nginx facilitate developers to extend the set of the web server features. The SSL/TLS asynchronous offload mode and the compression pipelining offload mode are implemented as dynamic modules in Nginx, named *ngx_ssl_engine_qat_module* and *ngx_http_qatzip_filter_module*, respectively.

Particularly, we extend the simple SSL engine setting scheme in Nginx to an SSL Engine Framework. As a new core framework in Nginx, the SSL Engine Framework adopts a hierarchical modular design to support different kinds of crypto accelerators and their custom configuration. A top-level *ngx_ssl_engine_module* provides the *ssl_engine* configuration block, along with basic data structures and interfaces. A core sub-module named *ngx_ssl_engine_core_module* provides the *use_engine* (specifying which engine to use) and *default_algorithm* (specifying which algorithms supplied by the engine should be used by default) configuration directives. The dynamic *ngx_ssl_engine_qat_module* is an engine-specific sub-module that provides the *qat_engine* configuration block and qat-specific configuration directives. It implements the interface functions defined by the core *ngx_ssl_engine_module*. If developers want to enable another crypto accelerator for SSL/TLS offloading, the only work is to refer to the *ngx_ssl_engine_qat_module* module and implement a new engine-specific module.

The *ngx_http_qatzip_filter_module* is a standard filter module that provides configuration for compression offloading. Developers can directly refer to this module to enable new compression accelerator for Nginx as long as a compression library similar to our pipelining compression library is provided.

An example Nginx configuration with QAT offloading for both SSL/TLS and compression is as follows:

```
load_module   modules/ngx_ssl_engine_qat_mod-
ule.so
load_module   modules/ngx_http_qatzip_filter_-
module.so
...
ssl_engine {
    use qat_engine;
    default_algorithm RSA,EC,DH,PKEY_CRYPTO;
    qat_engine {
        qat_offload_mode async;
        qat_notify_mode poll;
        qat_poll_mode heuristic;
        qat_heuristic_poll_asym_threshold 48;
        qat_heuristic_poll_sym_threshold 24;
    }
}
http {
    server {
        ...
        qatzip on;
        qatzip_comp_level 1;
        qatzip_sw_threshold 256;
        qatzip_chunk_size 64k;
        qatzip_stream_size 256k;
    }
}
```

5 Evaluation

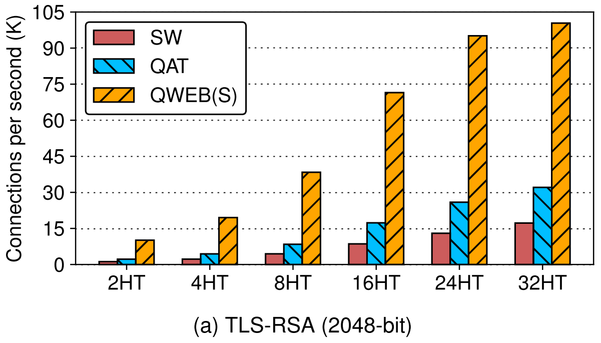
5.1 Experimental Setup

We established an experimental testbed with one tested server and one client server that were connected back-to-back via Intel XL710 40 GbE NICs. All servers were equipped with two 22-core Intel Xeon E5-2699 v4 CPU (hyper-threading enabled) and 64 GB RAM. We ran CentOS 7.3 with Linux Kernel 3.10 on them. QWEB was installed on the tested server with one Intel DH8970 PCIe QAT device.

The tested QWEB was based on Nginx 1.14.2, OpenSSL 1.1.0g and QAT Engine v0.5.37. Nginx supports a number of working modes, such as a web server, a reverse proxy server or a load balancer. Since all these modes share the same processing for SSL/TLS and HTTP compression, we only configured Nginx as an HTTPS web server to conduct the testing. In the tested server, multiple Nginx workers were configured to occupy the same number of dedicated hyper-threading (HT) cores.

Nginx uses the gzip format with deflate algorithm [67] for both dynamic and static compression. As mentioned above, deflate is a heavyweight compression algorithm with high compression ratio. Our evaluation targets the dynamic compression and the deflate algorithm was offloaded to QAT to validate the pipelining offload mode. The gzip compression level was set to one (default setting in Nginx) for both software and QAT cases.

The evaluation was conducted in terms of (1) SSL/TLS performance (i.e., only SSL/TLS offloaded and compression disabled) and (2) HTTP compression performance (i.e., only compression offloaded) and (3) performance of the



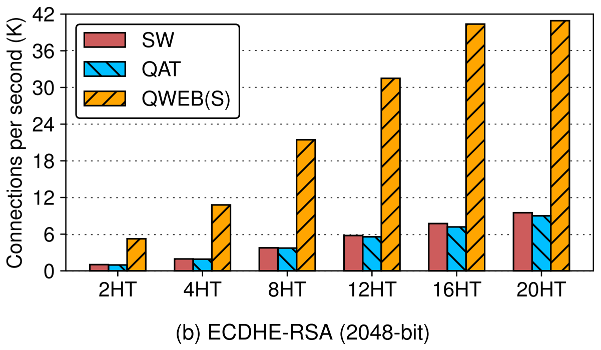


Fig. 7. Full handshake performance for two mainstream RSA-related cipher suites in the TLS 1.2 protocol.

combined offloading. Specifically, the following five configurations were used and compared:

- *SW*: software calculation for SSL/TLS related crypto operations and HTTP compression.
- *QAT*: using QAT with straight offload mode for SSL/TLS processing and HTTP compression.
- *QWEB(S)*: using QAT with SSL/TLS asynchronous offload mode to accelerate the HTTPS web server (HTTP compression disabled)
- *QWEB(C)*: using QAT with compression pipelining offload mode to accelerate the HTTPS web server (without SSL/TLS offloading)
- *QWEB*: using QAT with both the SSL/TLS asynchronous offload mode and the compression pipelining offload mode (i.e., the combined offloading).

Note that the tested DH8970 QAT device contains three independent QAT endpoints. We configured each QAT endpoint to provide 32 crypto instances and 32 compression instances [58]. In our evaluation, each Nginx worker process in the tested server was equipped with three crypto instances and/or three compression instances, which were evenly allocated from the three independent QAT endpoints. In this way, each Nginx worker can employ all parallel computation engines in the three QAT endpoints as QAT Engine is designed to submit crypto/compression requests to different QAT instances in a round-robin way.

5.2 SSL/TLS Performance

This subsection evaluates the pure SSL/TLS performance of QWEB with HTTP compression disabled. Three configurations were used and compared: SW, QAT and QWEB(S). The performance advantages of the *QWEB(S)* configuration over the *QAT* configuration are actually comprised of three parts: the asynchronous offload mode, the heuristic polling scheme and the kernel-bypass notification scheme. In general, the heuristic polling scheme provides an additional 10-20 percent performance enhancement and the kernel-bypass notification scheme provides an additional 5-10 percent performance enhancement (refer to our prior conference version [64] for the detailed measurement).

5.2.1 SSL/TLS Handshake

The SSL/TLS handshake performance was measured with the OpenSSL *s_time* tool. Each Nginx worker in the tested server was equipped with three QAT crypto instances. In the client server, 2000 *s_time* processes were simultaneously launched to establish SSL/TLS connections with the Nginx running in the tested server. We first evaluated the full handshake performance (i.e., establishing new SSL/TLS connections always) for mainstream cipher suites of the TLS 1.2 protocol: TLS-RSA, ECDHE-RSA and ECDHE-ECDSA.

The full handshake performance of TLS-RSA (2048-bit) is shown in Fig. 7a. Here ”2HT” indicates that two Nginx workers are running on two dedicated hyper-threading cores (belonging to the same physical core). It can be seen that the connections per second value increases linearly for all three configurations when the number of Nginx workers varies from 2 to 24. Taking the 8HT case as an example, the *SW* configuration gives 4.4k CPS. The straight offloading of RSA and PRF operations (i.e., the *QAT* configuration) improves the CPS by a factor of about two. In the *QWEB(S)* configuration, the SSL/TLS asynchronous offload mode fully utilizes both CPU and QAT resources, bringing a CPS boost. Along with the optimizations provided by the heuristic polling scheme and the kernel-bypass notification scheme, the *QWEB(S)* configuration achieves a final CPS up to 38.4k, which is a nearly **9x** CPS improvement over the software baseline. For the 32HT case, the *QWEB(S)* configuration provides about 100k CPS, which has achieved the upper limit of the DH8970 QAT card.

Fig. 7b shows the handshake performance of ECDHE-RSA (2048-bit), which involves two more ECC calculations compared to the TLS-RSA cipher suite. The OpenSSL default curve, NIST P-256, was used for the ECC computation. The *QAT* configuration shows no CPS improvement over the *SW* configuration due to the long blockings in the offload I/O. The *QWEB(S)* configuration with the SSL/TLS asynchronous offload mode and two optimizations improves the CPS by a factor of about **5.5** over the software baseline, with 16 Nginx workers to achieve the QAT upper limit (i.e., 40k CPS).

For another mainstream cipher suite ECDHE-ECDSA that involves three ECC calculations, we evaluated six NIST curves with four Nginx workers, as shown in Fig. 8. A striking phenomenon is that for the P-256 curve, the *SW* configuration provides an abnormal high CPS and outperforms the *QAT* configuration significantly. This is because the prime of the P-256 curve is ”Montgomery Friendly” and can be implemented into the Montgomery domain [68]. This feature makes the ECDSA (P-256) sign 2.33x faster than the traditional implementation. Nevertheless, QWEB enhances the CPS by nearly 70 percent compared to the special software

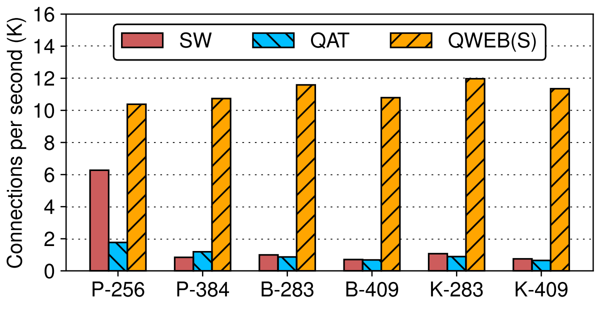


Fig. 8. Full handshake performance for ECDHE-ECDSA.

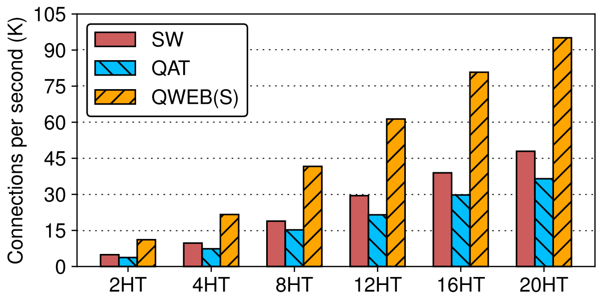


Fig. 9. Session resumption performance (full/abbreviated = 1:9).

implementation. It is worth noting that not all ECC prime curves can be optimized by the Montgomery domain. For the stronger P-384 curve, QWEB provides a **13x** CPS improvement over the software baseline. Moreover, according to SafeCurves [69], the widely-used P-256 curve has potential security risks. Therefore, we also evaluated another four counterpart curves: NIST binary curves (B-283 and B-409) and NIST koblitz curves (K-283 and K-409). For all these four curves, QWEB enhances the CPS by more than **11x** compared to the *SW* configuration.

Session resumption are widely used in real-life deployment to reduce the resource consumption at the server side. In the TLS 1.2 protocol, an abbreviated handshake for session resumption involves PRF calculations only. As mentioned above, session resumption compromises the protection afforded by forward secrecy and the lifetime of session IDs or tickets is typically restricted. Real-life traffic is a mixture of full and abbreviated handshakes and the ratio between them depends on the security policy. We evaluated the session resumption performance using an example ratio of 1:9 (i.e., 10 percent full handshakes) with the cipher suite ECDHE-RSA (2048-bit), as shown in Fig. 9. Compared to the *SW* configuration, QWEB improves the CPS by a factor more than *two*. Given a bigger percentage of full handshakes (i.e., more stringent security policy), a higher CPS enhancement can be achieved.

5.2.2 Secure Data Transfer

We used Apachebench (ab) to measure secure data transfer throughput with the requested file size varying from 4 KB to 1024 KB. The cipher suite for secure data transfer was AES128-SHA. In the tested server, eight Nginx workers were launched to employ 24 crypto instances and the keepalive setting was tuned to avoid the influence of SSL/TLS handshakes. In the client sever, 400 ab processes were configured to continuously request for a fixed file.

The experiments results are shown in Fig. 10. For the 4 KB case, QWEB only provides a slight higher throughput compared to the *SW* configuration because of the following two reasons. First, each request for a 4 KB file only triggers one encryption operation. Second, the offload benefit of a small 4 KB data block is largely offset by the offload overhead. As the requested file size increases, the performance benefits of QWEB becomes bigger because the data encryption for the file incurs more cipher operations at the server side. Taking the 128 KB case as an example, each request (i.e., one file) incurs eight (calculated by 128/16) cipher operations. The *QWEB(S)* configuration provides more than **2x** throughput improvement over the software baseline while the *QAT* configuration (i.e., the straight offload mode) shows no performance improvement and even behaves worse.

5.3 HTTP Compression Performance

This subsection evaluates the pure HTTP compression performance of QWEB without SSL/TLS offloading. Each Nginx worker was equipped with three QAT compression instances and the keepalive setting was tuned to avoid the influence of SSL/TLS handshakes. In the client sever, 400 ab processes were configured to continuously request for files or pages. Three configurations were used and compared: SW, QAT and QWEB(C).

5.3.1 Pipelining Parameter Selection

We first used experiments to determine proper values for the pipelining parameters: *qatzip_stream_size* and *qatzip_chunk_size*. In the tested server, one Nginx worker was launched on a dedicate core. The ab processes in the client server were configured to request for a fixed file or page.

The first experiment tested a 512 KB file truncated from the 597 KB book2 in the large Calgary Corpus [70], which is a benchmark corpus for lossless compression evaluation.

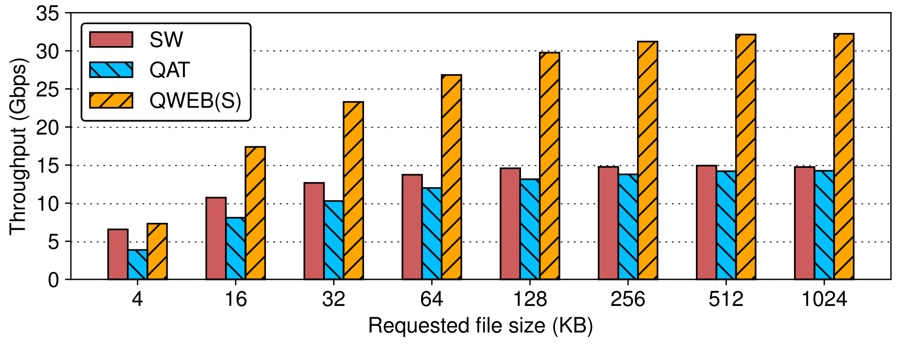
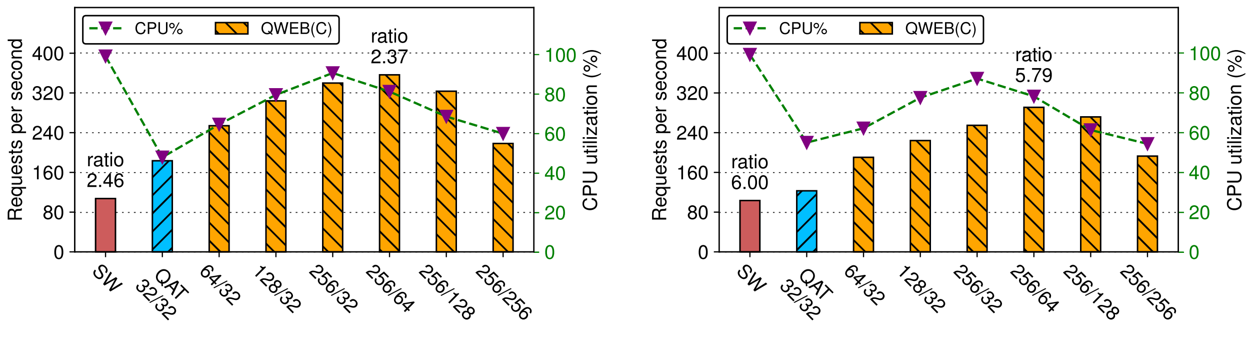


Fig. 10. Secure data transfer throughput with AES128-SHA cipher suite and varied requested file size.



(a) 512KB file derived from Calgary Corpus (b) 1024KB page derived from a news website

Fig. 11. Compression performance with varied pipelining offload parameters while requesting for a fixed file or page.

The experiment results are shown in Fig. 11a. The *SW* configuration can only process 107 requests per second (RPS) with an entirely used CPU core. The label "QAT 32/32" denotes the pipelining offload mode with both *qatzip_stream_size* and *qatzip_chunk_size* set to 32 KB. That's to say, once Nginx prepares 32 KB content (default buffer size), this 32 KB content is sent to QAT as a request without accumulation. Therefore, this configuration is equivalent to the QAT configuration with straight offload mode. The RPS in this configuration is 183, which is a 70 percent enhancement compared to the software baseline. However, the 48 percent CPU utilization implies that the offload performance is impeded by the frequent blockings and still has room for improvement.

As the *qatzip_stream_size* varies from 32 KB to 256 KB, more pipelines are used, leading to the gradual increase of both RPS and CPU utilization. For the case of *qatzip_stream_size* set to 256 KB, eight concurrent pipelines (calculated by 256/32) are employed to amortize blocking penalty, with the CPU utilization rising to 91 percent. Furthermore, we increase the *qatzip_chunk_size* for comparison. It can be observed that the 64 KB case for *qatzip_chunk_size* behaves slightly better than the 32 KB case on both throughput and CPU utilization due to the reduced offload overhead (i.e., the number of requests/responses). However, the further increase of *qatzip_chunk_size* to 256 KB degrades the concurrency and impedes the throughput. Finally, we select **256 KB** and **64 KB** as the recommended setting for the two pipelining parameters: *qatzip_stream_size* and *qatzip_chunk_size*, respectively. And its compression ratio (2.37) is only slightly lower than that (2.46) of the software case.

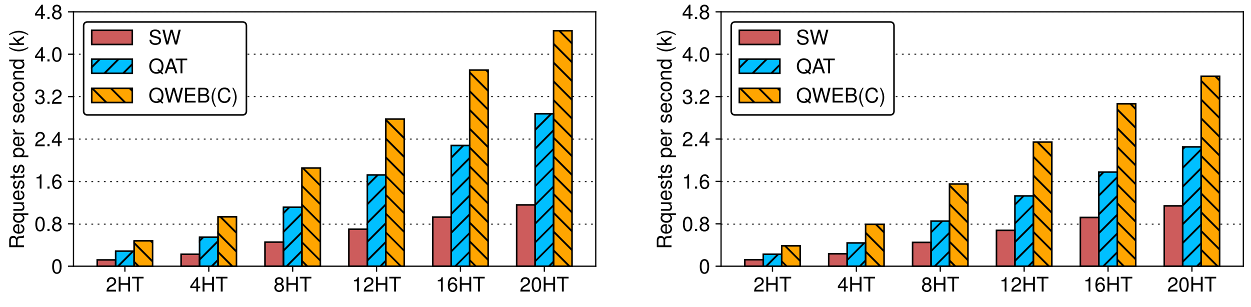
Considering that Calgary Corpus is a general benchmark and may not be able to catch the characteristics of the web applications, we also tested a 1024 KB page truncated from the real-life 1101 KB front page of CNN news website, as shown in Fig. 11b. Still, the "256 KB/64 KB" cominbation for *qatzip_stream_size* and *qatzip_chunk_size* behaves best in throughput with acceptable CPU utilization and compression ratio.

To evaluate the scalability of the compression pipelining offload mode with the above recommended parameters (i.e., *qatzip_stream_size* set to 256 KB and *qatzip_chunk_size* set to 64 KB in the QWEB(C) configuration), we varied the number of Nginx workers from 2 to 20, as shown in Fig. 12. Similarly, "2HT" indicates that two Nginx workers are running on two dedicated hyper-threading cores (belonging to the same physical core). We can observe that the RPS value increases linearly for all configurations when the number of Nginx workers increases gradually. In general, the performance advantage of QWEB over the software baseline is about **4x** while the 512 KB file is requested and about **3.5x** while the 1024 KB page is requested.

5.3.2 Real-Life Web Pages

After determining the proper pipelining parameters, we further evaluated the HTTP compression performance using real-life web pages with different sizes. We chose five front pages from popular news websites, as listed in Table 3. Eight Nginx workers were launched in the tested server and the ab processes in the client server were configured continuously request for a fixed page.

Fig. 13 presents the comprehensive experiment results in terms of completion time (throughput), compression ratio and CPU utilization. For the small abc page (first 5,000 bytes



(a) 512KB file derived from Calgary Corpus (b) 1024KB page derived from a news website

Fig. 12. Compression performance scalability with varied number of nginx workers while requesting for a fixed file or page.

TABLE 3
Five News Front Pages With Different Sizes

News Website	Denotation	Web Page Size
ABC News	abc	(first) 5,000 bytes
Shanghai Daily	shine	49,842 bytes
The Statesman	statesman	145,047 bytes
Times of India	india	269,914 bytes
CNN International	cnn	1,127,194 bytes

of the official page), the *QWEB(C)* configuration shows no advantage over the *SW* configuration because the small 5,000-byte size means only one compression offload job per HTTP request (i.e., no concurrent pipelines) and the offload benefit is also small, largely offset by the offload overhead. As the increase of the page size, the performance advantage of QWEB becomes more and more obvious. For the cnn page that can fully benefit from the compression pipelining offload mode, QWEB only takes 66.5s to complete 10,0000 requests, which is a 72 percent reduction in completion time compared to the 237.3s of the software baseline. In other words, QWEB improves the RPS throughput by a factor of about **3.5** over the software compression. Also, the obvious performance gap between the the *QAT* configuration (i.e., the straight offload mode) and the *QWEB(C)* configuration proves the necessity of the compression pipelining offload mode. Additionally, the CPU utilization of the software case is always 100 percent while the CPU utilization of QWEB decreases from 94.8 to 81.7 percent as the increase of the page size.

The 3.5x compression performance enhancement with 81.7 percent CPU utilization proves not only the superior performance of the pipelining offload mode but also its small performance gap from an asynchronous one. In other words, even if we make efforts to enable an asynchronous offload mode that can fully utilize the CPU resources (i.e., 100 percent utilization), the extra performance advantage is limited compared to the pipelining offload mode with above 80 percent CPU utilization. Moreover, the incurred overhead of context switches among asynchronous offload jobs may further offset the advantage.

For all five news front pages, the *QWEB(C)* configuration and the *SW* configuration achieve similar compression ratio, with a deviation within 4 percent, as shown in Fig. 13b. Software (i.e., the zlib library) compresses the full content into a deflate stream consisting of a series of blocks, and does not try to determine when it might be beneficial to end a block earlier. QAT compresses each offloaded chunk (e.g., 64 KB) of the full content into one deflate block and the final compression result is also a series of deflate blocks. This is why sometimes the *SW* configuration provides a higher compression ratio but other times the *QWEB(C)* configuration may achieve better. Although the QWEB(C) configuration gives a slightly lower compression ratio in some cases, it is acceptable in consideration of the huge performance advantage.

5.4 Performance of Combined Offloading

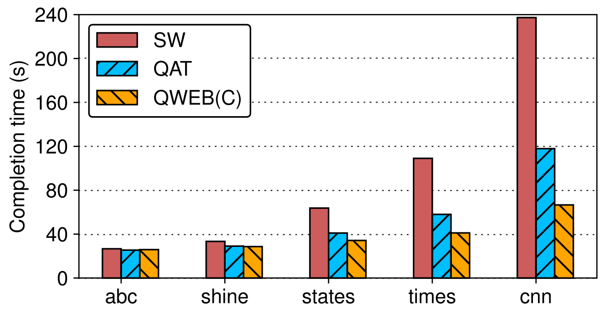
This subsection evaluates the combined offloading of SSL/TLS processing (mainly data encryption) and HTTP compression. The experimental setting was basically the same as that of the above testing for real-life web pages. Each Nginx worker was equipped with three QAT crypto instances and three QAT compression instances, indicating that the eight Nginx workers totally employed 48 QAT instances.

As shown in Fig. 14, compared to the above testing that offloads compression only, the combined offloading of QWEB shows impressive performance enhancement over the software baseline for all five web pages with different sizes. In general, QWEB improves the RPS throughput by a factor of more than *five* and reduces the completion time by more than 80 percent compared to the *SW* configuration. The reason that the small page case in the *QWEB* configuration shows a high performance advantage as well is that there are a number of async crypto offload jobs are being calculated by the QAT accelerator when a blocking compression offload job is conducted. Therefore, the blocking penalty is further atomized by the concurrent crypto offload jobs.

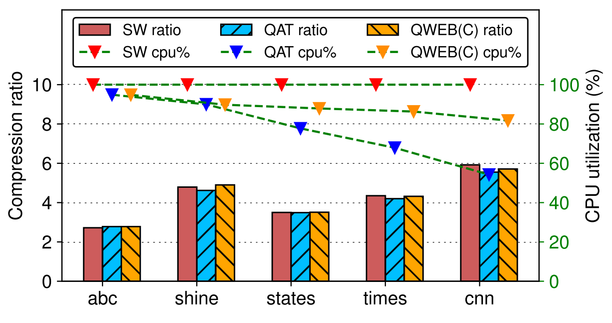
In contrast, the *QAT* configuration (i.e., the straight offload mode) only can reduce the completion time by about half because of the frequent blockings in the crypto/compression offload I/O. Another phenomenon resulting from the frequent blockings is that the CPU utilization of the *QAT* configuration has an obvious drop as the increased size of the requested page, as shown in Fig. 14b. For the QWEB configuration, since both compression and encryption are offloaded to the QAT accelerator, its CPU utilization is lower than that of the above testing that offloads compression only.

6 Related Work

SSL/TLS Acceleration. Prior studies that accelerate SSL/TLS processing can be classified into three categories: software acceleration, CPU-level improvement and hardware offloading.



(a) Completion Time for 10,0000 Requests



(b) Compression Ratio and CPU Utilization

Fig. 13. HTTP compression performance without SSL/TLS offloading for five news front pages with different sizes.

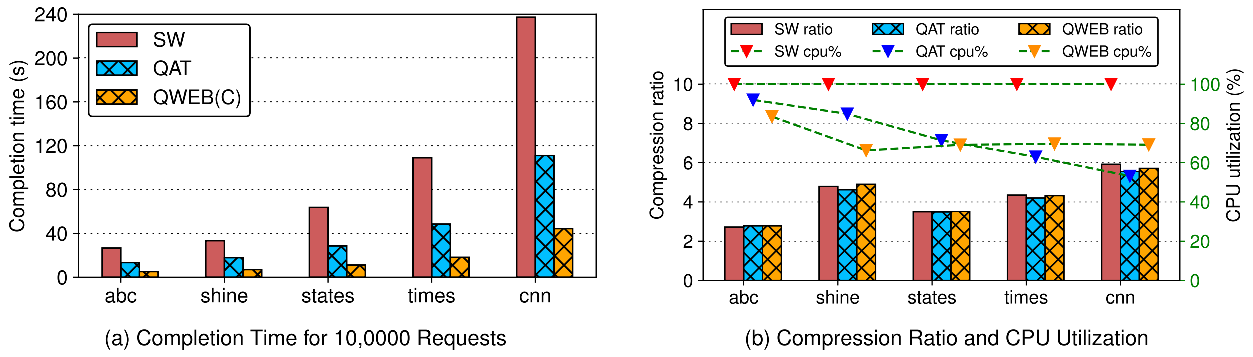


Fig. 14. Combined offloading of SSL/TLS processing and HTTP compression for five news front pages with different sizes.

Shacham *et al.* [71] presented an algorithmic approach, batching the SSL handshakes on the web server with Batch RSA, to speed up SSL. Boneh *et al.* [72] described three fast variants of RSA (Batch RSA, Multi-factor RSA and Rebalanced RSA) and analyzed their security issues. A client-side caching of handshake information is proposed in [73] to reduce the network overhead. Castelluccia *et al.* [74] examined a technique for re-balancing RSA-based client/server handshakes, which puts more work on clients to achieve better SSL performance on the server.

AES instruction set, an extension to the x86 instruction set architecture, has been integrated into many processors to improve the speed of encryption and decryption using AES [75]. Kounavis *et al.* [76] presented a set of processor instructions along with software optimizations that can be used to accelerate end-to-end encryption and authentication. In [77], a novel constant run-time approach was proposed to implement fast modular exponentiation on IA processors.

Researchers have enabled SSL/TLS related crypto algorithms (e.g., RSA, AES and HMAC) on different types of hardware accelerators, including GPU [5], [16], [17], [18], FPGA [22], [23], [24] and Intel Xeon Phi processor [13]. The main focus of these works is to program the general-purpose accelerators to conduct efficient crypto calculation.

Compression Acceleration. The primary compression acceleration approach is the hardware acceleration, which has been well studied in the literature by using general-purpose accelerators.

Patel *et al.* [20] proposed a parallel algorithm and implemented a bzip2-like lossless data compression scheme on GPU architecture. Their implementation enabled the GPU to become a co-processor of data compression, which lightened the computing burden of CPU by using idle GPU cycles. Ozsoy *et al.* [19] presented a pipelined parallel LZSS compression algorithm for GUGPU. Abdelfattah *et al.* [50] utilized the Open Computing Language to implement high-speed gzip compression on an FPGA, which achieved higher throughput over standard compression benchmark, with equivalent compression ratio. Fowers *et al.* [25] detailed a scalable fully pipelined FPGA accelerator that performs LZ77 compression and static Huffman encoding at rates up to 5.6 GB/s.

Comparison. These above studies mainly concentrated on the accelerator programming to enable efficient calculation of the needed crypto or compression algorithms, without paying much attention to the offload I/O. By contrast, the proposed QWEB is based on the QAT ASIC, which inherently provides high-performance and energy-efficient calculation for crypto and compression. Our work focuses on the offload I/O of the event-driven web architecture and optimizes the straight offload mode to allow concurrent offload tasks so as to achieve high-performance SSL/TLS and compression acceleration.

7 Discussion and Future Work

Hardware/Software Switch. Each offload job incurs offload overhead for the processing of QAT request/response and PCIe transactions. Consequently, for the crypto or compression offloading of very small data block, the offload benefit may be totally offset by the offload overhead, making software calculation become a better choice. QWEB provides customizable hardware/software switch (through Nginx configuration file) for both SSL/TLS and compression offloading. Note that lightweight or heavyweight crypto/compression algorithm may have a considerable influence on the performance trade-off between offloading or not. For example, the default compression scheme in QWEB is the costly gzip/deflate. If a lightweight compression algorithm is applied in the future, the threshold for hardware/software switch should be increased properly according to empirical tests.

Software Fallback. It is possible that hardware failure occurs while performing SSL/TLS or compression offloading, which is actually a key concern in real-life deployment. QWEB has provided robust software fallback mechanism to switch to software mode for crypto or compression calculation once detecting hardware malfunction. The configuration of the software fallback mechanism is through the *qat_sw_fallback* directive (within the *ssl_engine* block) for SSL/TLS offloading and the *qatzip_sw* directive for compression offloading.

Hybrid Approach. The proposed QWEB is based on the userspace polling to retrieve QAT responses, which is able to maximize system performance under high offload load because one userspace polling operation has much less overhead than one kernel-based interrupt operation [61]. On the other hand, it is inevitable that the polling scheme, no matter the basic periodical polling or the proposed heuristic polling, may incur ineffective polling operations (i.e., no QAT response retrieved because of unfinished requests or no pending request at all), especially under low offload load. Therefore, in consideration of energy saving, a hybrid approach can be studied to reap the strength of both polling mode and interrupt mode by performing proper switch between them (e.g., according to the real-time or predicted offload load).

Chained Offloading and SmartNIC. The QAT accelerator supports crypto and compression offloading simultaneously. A natural idea is that whether we can use a chained offloading of compression and encryption, i.e., one offload request to make the QAT accelerator to perform first compression and then encryption on the submitted data chunk. In this way, the total offload overhead (e.g., DMA transfers) for the HTTPS web server could be reduced by half. Furthermore, if a SmartNIC [3] can provide crypto/compression service and one offload request can make the SmartNIC to perform compression, encryption and transmission in sequence, the number of DMA transfers between application and hardware device can be reduced to only one. However, the enabling of the chained offloading or SmartNIC integration requires not only hardware support, but also a re-engineering of the web software architecture. We take this as one of our future work.

8 Conclusion

In this paper, we proposed and designed QWEB, a comprehensive offload solution for the popular event-driven web architecture to achieve high-performance SSL/TLS and compression acceleration. QWEB re-engineered the SSL/TLS software stack to provide SSL/TLS asynchronous offload mode and took advantage of the stream feature of HTTP compression to provide compression pipelining offload mode. These two novel mode allows concurrent offload tasks from a single application process/thread so as to increase the utilization rate of both CPU and accelerator resources. We have implemented the QWEB prototype based on the widely-deployed Nginx. The comprehensive performance evaluation demonstrated that QWEB outperforms the software baseline by a significant margin for both SSL/TLS handshake and secure data transfer.

Acknowledgments

The authors would like to thank the anonymous referees for their valuable comments. This article extends a prior conference version [64] that appeared in the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '19). This work was supported in part by the National Natural Science Foundation of China (No. 61972245), the National Key Research and Development Program of China (No. 2016YFB1000502), and the National Science Fund for Distinguished Young Scholars (No. 61525204).

References

[1] M. Khazraee, L. Zhang, L. Vega, and M. B. Taylor, "Moonwalk: NRE optimization in ASIC clouds," in *Proc. 21st Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2017, pp. 511–526.

[2] A. M. Caulfield *et al.*, "A cloud-scale acceleration architecture," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2016, pp. 7:1–7:13.

[3] D. Firestone *et al.*, "Azure accelerated networking: Smartnics in the public cloud," in *Proc. 15th USENIX Symp. Netw. Syst. Des. Implementation*, 2018, pp. 51–66.

[4] Y. Liu, J. Wang, and S. Swanson, "Griffin: Uniting CPU and GPU in information retrieval systems for intra-query parallelism," in *Proc. 23rd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2018, pp. 327–337.

[5] K. Jang, S. Han, S. Han, S. B. Moon, and K. Park, "SSLShader: Cheap SSL acceleration with commodity processors," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 1–14.

[6] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun.*, 2017, pp. 15–28.

[7] T. Dierks, "The transport layer security (TLS) protocol version 1.2," 2008. [Online]. Available: https://tools.ietf.org/html/rfc5246

[8] S. Helme, "Top 1 million analysis - March 2020, " 2020. Accessed: May 2020. [Online]. Available: https://scotthelme.co.uk/top-1-million-analysis-march-2020/

[9] R. Fielding *et al.*, "Hypertext transfer protocol–http/1.1," 1999. [Online]. Available: https://tools.ietf.org/html/rfc2616

[10] Usage statistics of compression for websites. 2020. Accessed: May 2020. [Online]. Available: https://w3techs.com/technologies/details/ce-compression/all/all

[11] C. Coarfa, P. Druschel, and D. S. Wallach, "Performance analysis of TLS web servers," *ACM Trans. Comput. Syst.*, vol. 24, no. 1, pp. 39–69, 2006.

[12] D. Naylor *et al.*, "The cost of the "S" in HTTPs," in *Proc. 10th ACM Int. Conf. Emerg. Netw. Experiments Technol.*, 2014, pp. 133–140.

[13] S. Yao and D. Yu, "PhiOpenSSL: Using the xeon phi coprocessor for efficient cryptographic calculations," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2017, pp. 565–574.

[14] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy, "Potential benefits of delta encoding and data compression for HTTP," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 4, pp. 181–194, 1997.

[15] E. Zohar and Y. Cassuto, "Automatic and dynamic configuration of data compression for web servers," in *Proc. 28th Large Installation Syst. Admin. Conf.*, 2014, pp. 106–117.

[16] J. Yang and J. Goodman, "Symmetric key cryptography on modern graphics hardware," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, 2007, pp. 249–264.

[17] O. Harrison and J. Waldron, "Practical symmetric key cryptography on modern graphics hardware," in *Proc. 17th USENIX Secur. Symp.*, 2008, pp. 195–210.

[18] R. Szerwinski and T. Güneysu, "Exploiting the power of GPUs for asymmetric cryptography," in *Proc. 10th Int. Workshop Cryptographic Hardware Embedded Syst.*, 2008, pp. 79–99.

[19] A. Ozsoy, M. Swany, and A. Chauhan, "Pipelined parallel LZSS for streaming data compression on GPGPUs," in *Proc. 18th Int. Conf. Parallel Distrib. Syst.*, 2012, pp. 37–44.

[20] R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens, "Parallel lossless data compression on the GPU," in *Proc. Innovative Parallel Comput.*, 2012, pp. 1–9.

[21] B. Zhou, H. Jin, and R. Zheng, "A high speed lossless compression algorithm based on CPU and GPU hybrid platform," in *Proc. 13th Int. Conf. Trust Secur. Privacy Comput. Commun.*, 2014, pp. 693–698.

[22] T. Isobe, S. Tsutsumi, K. Seto, K. Aoshima, and K. Kariya, "10 Gbps implementation of TLS/SSL accelerator on FPGA," in *Proc. 18th Int. Workshop Quality Service*, 2010, pp. 1–6.

[23] M. I. Soliman and G. Y. Abozaid, "FPGA implementation and performance evaluation of a high throughput crypto coprocessor," *J. Parallel Distrib. Comput.*, vol. 71, no. 8, pp. 1075–1084, 2011.

[24] Z.-U.-A. Khan and M. Benaissa, "Throughput/area-efficient ECC processor using montgomery point multiplication on FPGA," *IEEE Trans. Circuits Syst. II: Express Briefs*, vol. 62, no. 11, pp. 1078–1082, Nov. 2015.

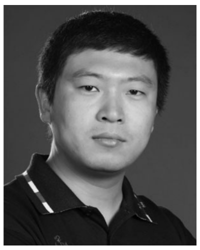
[25] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, "A scalable high-bandwidth architecture for lossless compression on FPGAs," in *Proc. 23rd Int. Symp. Field-Programmable Custom Comput. Machines*, 2015, pp. 52–59.

[26] W. Qiao *et al.*, "High-throughput lossless compression on tightly coupled CPU-FPGA platforms," in *Proc. 26th Int. Symp. Field-Programmable Custom Comput. Machines*, 2018, pp. 37–44.

[27] W. Reese, "NGINX: The high-performance web server and reverse proxy," *Linux J.*, vol. 2008, no. 173, 2008, Art. no. 2.

[28] Haproxy - The reliable, high performance TCP/HTTP load balancer. 2019. Accessed: Aug. 2019. [Online]. Available: http://www.haproxy.org/

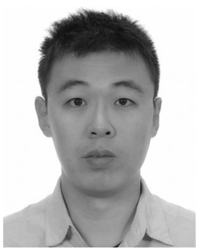
[29] Squid: Optimising web delivery. 2019. Accessed: Aug. 2019. [Online]. Available: http://www.squid-cache.org/
[30] Intel® quickassist technology (intel® QAT). 2019. Accessed: Aug. 2019. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html
[31] CloudFlare boosts performance and stability for its millions of websites with nginx. 2018. Accessed: Aug. 2019. [Online]. Available: https://www.NGINX.com/success-stories/cloudflare-boosts-performance-sta bility-millions-websites-with-nginx/
[32] Usage of web servers broken down by ranking. 2019. Accessed: Aug. 2019. [Online]. Available: https://w3techs.com/technologies/cross/web_server/ranking
[33] Intel® quickassist technology (QAT) async mode NGINX. 2019. Accessed: Aug. 2019. [Online]. Available: https://github.com/intel/asynch_mode_nginx
[34] tengine QAT SSL. 2019. Accessed: Aug. 2019. [Online]. Available: http://tengine.taobao.org/document/tengine_qat_ssl.html
[35] Working together to build a high efficiency CDN system for HTTPS, Intel and Wangsu, 2018. [Online]. Available: https://01.org/sites/default/files/downloads/intelr-quickassist-technology/i12036-casestudy-intelqatcdnaccelerationen337190-001us.pdf
[36] Transport layer security. 2019. Accessed: Aug. 2019. [Online]. Available: https://en.wikipedia.org/wiki/Transport_Layer_Security
[37] Pseudorandom function family. 2019. Accessed: Aug. 2019. [Online]. Available: https://en.wikipedia.org/wiki/Pseudo random_function_family
[38] R. Canetti, S. Halevi, and J. Katz, "A forward-secure public-key encryption scheme," in *Proc. Int. Conf. Theory Appl. Cryptographic Techn.*, 2003, pp. 255–271.
[39] Elliptic-curve Diffie-Hellman. 2019. Accessed: Aug. 2019. [Online]. Available: https://en.wikipedia.org/wiki/Elliptic-curve_Diffie-Hellman
[40] Elliptic-curve cryptography. 2019. Accessed: Aug. 2019. [Online]. Available: https://en.wikipedia.org/wiki/Elliptic-curve_cryptography
[41] E. Rescorla, "The transport layer security (TLS) protocol version 1.3," 2018. [Online]. Available: https://tools.ietf.org/html/rfc8446
[42] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe, "A comprehensive symbolic analysis of TLS 1.3," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 1773–1788.
[43] H. Krawczyk and P. Eronen, "HMAC-based extract-and-expand key derivation function (HKDF)," 2010. [Online]. Available: https://tools.ietf.org/html/rfc5869
[44] A. Rawdat, "Testing the performance of NGINX and NGINX plus web servers," 2017. Accessed: Aug. 2019. [Online]. Available: https://www.nginx.com/blog/testing-the-performance-of-nginx-and-nginx-p lus-web-servers/
[45] D. Springall, Z. Durumeric, and J. A. Halderman, "Measuring the security harm of TLS crypto shortcuts," in *Proc. Internet Meas. Conf.*, 2016, pp. 33–47.
[46] S. Souders, "High performance web sites," *ACM Queue*, vol. 6, no. 6, pp. 30–37, 2008.
[47] L. P. Deutsch, "Deflate compressed data format specification version 1.3," 1996. [Online]. Available: https://tools.ietf.org/html/rfc1951
[48] L. P. Deutsch, " GZIP file format specification version 4.3," 1996. [Online]. Available: https://tools.ietf.org/html/rfc1952
[49] Y. Chen, A. Ganapathi, and R. H. Katz, "To compress or not to compress-compute vs. IO tradeoffs for MapReduce energy efficiency," in *Proc. 1st ACM SIGCOMM Workshop Green Netw.*, 2010, pp. 23–28.
[50] M. S. Abdelfattah, A. Hagiescu, and D. Singh, "To compress or not to 1472 compress-compute vs. IO tradeoffs for MapReduce energy 1473 efficiency on a chip: High performance lossless data compression on FPGAs using OpenCL," in *Proc. Int. Workshop OpenCL*, 2014, pp. 4:1–4:9.
[51] B. Erb, "Concurrent programming for scalable web architectures," Universität Ulm, 2012. [Online]. Available: http://dx.doi.org/10.18725/OPARU-2423
[52] The apache HTTP server project. 2019. Accessed: Aug. 2019. [Online]. Available: https://httpd.apache.org/
[53] O. Garrett, "NGINX vs. apache: Our view of a decade-old question," 2015. Accessed: Aug. 2019. [Online]. Available: https://www.nginx.com/blog/nginx-vs-apache-our-view/
[54] L. Technologies, "Event-driven vs. process-based web servers," 2018. Accessed: Aug. 2019. [Online]. Available: https://www.litespeedtech.com/products/litespeed-web-server/features/ev ent-driven-architecture
[55] Epoll. 2019. Accessed: Aug. 2019. [Online]. Available: https://en.wikipedia.org/wiki/Epoll
[56] Kqueue. 2019. Accessed: Aug. 2019. [Online]. Available: https://en.wikipedia.org/wiki/Kqueue
[57] Programming intel® quickassist technology hardware accelerators for optimal performance, Intel, 2015. [Online]. Available: https://01.org/sites/default/files/page/332125_002_0.pdf
[58] Intel quickassist technology software for linux: Programmers guide - hardware version 1.7, Intel, 2020. [Online]. Available: https://01.org/sites/default/files/downloads/336210-012-intel-qat-swpg.pdf
[59] OpenSSL: Cryptography and SSL/TLS toolkit. 2019. Accessed: Aug. 2019. [Online]. Available: https://www.openssl.org/
[60] Intel® quickassist technology (QAT) openssl engine. 2019. Accessed: Aug. 2019. [Online]. Available: https://github.com/intel/QAT_Engine
[61] Intel® quickassist technology performance optimization guide, Intel, 2014. [Online]. Available: https://01.org/sites/default/files/page/330687_qat_perf_opt_guide_rev_1.0.pdf
[62] File descriptor. 2019. Accessed: Aug. 2019. [Online]. Available: https://en.wikipedia.org/wiki/File_descriptor
[63] zlib: A massively spiffy yet delicately unobtrusive compression library. 2017. Accessed: Aug. 2019. [Online]. Available: https://www.zlib.net/
[64] X. Hu *et al.*, "QTLS: High-performance TLS asynchronous offload framework with intel® quickassist technology," in *Proc. 24th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2019, pp. 158–172.
[65] Fiber (computer science). 2019. Accessed: Aug. 2019. [Online]. Available: https://en.wikipedia.org/wiki/Fiber_(computer_science)
[66] Boost.fiber 1.68.0 overview. 2018. Accessed: Aug. 2019. [Online]. Available: https://www.boost.org/doc/libs/1_68_0/libs/fiber/doc/html/fiber/overvie w.html
[67] Module ngx_http_gzip_module. 2019. Accessed: Aug. 2019. [Online]. Available: http://nginx.org/en/docs/http/ngx_http_gzip_module.html
[68] S. Gueron and V. Krasnov, "Fast prime field elliptic-curve cryptography with 256-bit primes," *J. Cryptographic Eng.*, vol. 5, no. 2, pp. 141–151, 2015.
[69] D. J. Bernstein and T. Lange, "SafeCurves: Choosing safe curves for elliptic-curve cryptography," 2017. Accessed: Aug. 2019. [Online]. Available: https://safecurves.cr.yp.to/
[70] Calgary corpus. 2019. Accessed: Aug. 2019. [Online]. Available: http://www.data-compression.info/Corpora/CalgaryCorpus/
[71] H. Shacham and D. Boneh, "Improving SSL handshake performance via batching," in *Proc. Cryptographer's Track RSA Conf.*, 2001, pp. 28–43.
[72] D. Boneh and H. Shacham, "Fast variants of RSA," *CryptoBytes*, vol. 5, no. 1, pp. 1–9, 2002.
[73] H. Shacham, D. Boneh, and E. Rescorla, "Client-side caching for TLS," *ACM Trans. Inf. Syst. Secur.*, vol. 7, no. 4, pp. 553–575, 2004.
[74] C. Castelluccia, E. Mykletun, and G. Tsudik, "Improving secure server performance by re-balancing SSL/TLS handshakes," in *Proc. ACM Symp. Inf. Comput. Commun. Secur.*, 2006, pp. 26–34.
[75] AES instruction set. 2019. Accessed: Aug. 2019. [Online]. Available: https://en.wikipedia.org/wiki/AES_instruction_set
[76] M. E. Kounavis, X. Kang, K. Grewal, M. Eszenyi, S. Gueron, and D. Durham, "Encrypting the Internet," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun.*, 2010, pp. 135–146.
[77] V. Gopal, J. Guilford, E. Ozturk, W. Feghali, G. Wolrich, and M. Dixon, "Fast and constant-time implementation of modular exponentiation," *Embedded Syst. Commun. Secur.*, 2009.



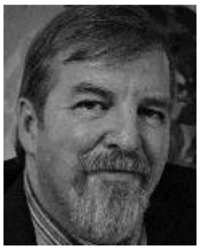
Jian Li (Member, IEEE) received the PhD degree in computer science from the Institut National Polytechnique de Lorraine (INPL) - Nancy, France, in 2007. He is an associate professor with the School of Software, Shanghai Jiao Tong University, China. He is a member of ACM and CCF. His research interests include virtualization, networking system and cloud computing.

Xiaokang Hu received the BS degree in computer science and technology from Nanjing University, China, in 2015. He is working toward the PhD degree majoring in the computer science and technology at Shanghai Jiao Tong University, China. He studies in the Shanghai Key Laboratory of Scalable Computing and Systems, guided by Prof. Haibing Guan. His research interests include virtualization, cloud computing and heterogeneous system.

David Qian received the bachelor's degree in mechatronics from Jiangsu University, China, in 2002. He is a software engineer of Intel Data Center Group. His research interests including data compression, networking system, and edge computing.



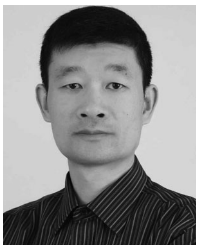
Changzheng Wei received the master's degree in microelectronics from the Graduate School of Chinese Academy of Sciences, China, in 2011. He is a software engineer with Intel Data Center Group. His research interests include computer architecture, applied cryptography, and network security.



Gordon McFadden received the bachelor's degree of computer science from the Technical University in Nova Scotia (nor Dalhousie University), Halifax, Nova Scotia, Canada. He is software architect working with Intel Data Platforms Group, currently focused on compression solutions for QuickAssist Technology.

Brian Will received the BEng degree in electronic engineering from Dublin City University, Ireland. He is a software architect with Intel Data Center Group. His Research interests include network security protocols, data compression algorithms, network inspection/filtering techniques and cloud networking.

Ping Yu received the master's degree in computer science from Southeast University, China, in 2003. He is a software engineer with Intel Data Center Group. His research interests include network security, data encryption, cloud computing, virtualization, and hardware acceleration technologies.



Weigang Li received the master's degree in computer science from Fudan University, China, in 2002. He is a software architect with Intel Data Center Group. His research interests include network security, data compression, storage workload optimization and hardware acceleration technologies.

Haibing Guan (Member, IEEE) received the PhD degree in computer science from the Tongji University, China, in 1999. He is a full professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. He is a member of ACM and CCF. His research interests include computer architecture, distrusted computing, virtualization, cloud computing, and system security.