This article has been accepted for publication in IEEE Computer Architecture Letters. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/LCA.2025.3534831

Hardware-accelerated Kernel-Space Memory Compression Using Intel® QAT

Qirong Xia†, Houxiang Ji†, *Student Member, IEEE*, Yang Zhou, and Nam Sung Kim, *Fellow, IEEE*

***Abstract*—Data compression has been widely used by datacenters to decrease the consumption of not only the memory and storage capacity but also the interconnect bandwidth. Nonetheless, the CPU cycles consumed for data compression notably contribute to the overall datacenter taxes. To provide a cost-efficient data compression capability for datacenters, Intel has introduced QuickAssist Technology (QAT), a PCIe-attached data-compression accelerator. In this work, we first comprehensively evaluate the compression/decompression performance of the latest *on-chip* QAT accelerator and then compare it with that of the previous-generation *off-chip* QAT accelerator. Subsequently, as a compelling application for QAT, we take a Linux memory optimization kernel feature: compressed cache for swap pages (`zswap`), re-implement it to use QAT efficiently, and then compare the performance of QAT-based `zswap` with that of CPU-based `zswap`. Our evaluation shows that the deployment of CPU-based `zswap` increases the tail latency of a co-running latency-sensitive application, Redis by 3.2-12.1×, while that of QAT-based `zswap` does not notably increase the tail latency compared to no deployment of `zswap`.**

***Index Terms*—Data Compression, Hardware Accelerator, Operating System.**

I. On-Chip QAT Accelerator

Intel QAT was initially introduced as an off-chip PCIe-attached accelerator [1] and is now integrated into Intel 4th-generation Xeon Scalable Processors, codenamed Sapphire Rapids (SPR) as an on-chip accelerator. The on-chip QAT retains a similar hardware architecture to its off-chip predecessor but features various new SoC-level features, enabling more versatile deployment and easier usage in datacenters.

Hardware Architecture. SPR is the first datacenter CPU architecture to integrate diverse on-chip accelerators directly with the CPU through a cache-coherent interconnect. Fig. 1 compares the differences between the off-chip and on-chip QAT connections to the CPU, as well as the hardware and software stack for the on-chip QAT accelerator. The off-chip QAT accelerator is exposed to the host CPU as a PCIe Endpoint (EP) device, connecting through the PCIe Upstream Port (USP) and Downstream Port (DSP). In contrast, the on-chip QAT accelerator is exposed as a PCIe-compatible Root Complex Integrated Endpoint (RCiEP) device, fully compatible with the standard PCIe configuration mechanism; unless specified otherwise, QAT refers to the on-chip accelerator in this paper. The host CPU can submit work descriptors and update configurations of QAT through the I/O fabric (between the cache-coherent interconnect and on-chip accelerators), and QAT can access host memory through the same interface. Ring pairs facilitate coordination between the host CPU and QAT and each pair consists of a transmission (TX) ring where the host submits work descriptors and a receiving (RX) ring where QAT writes status updates and results back to the host, respectively. The host CPU can configure each ring, such as base address and number of entries, via Configuration Status Registers (CSRs) accessed through Memory-Mapped I/O (MMIO). The work descriptors from the TX ring are arbitrated and dispatched to appropriate QAT engines based on the tasks specified in the descriptors and the status of the engines. Upon completing the tasks, the engines write completions back to the RX ring, where they await processing by the host CPU.

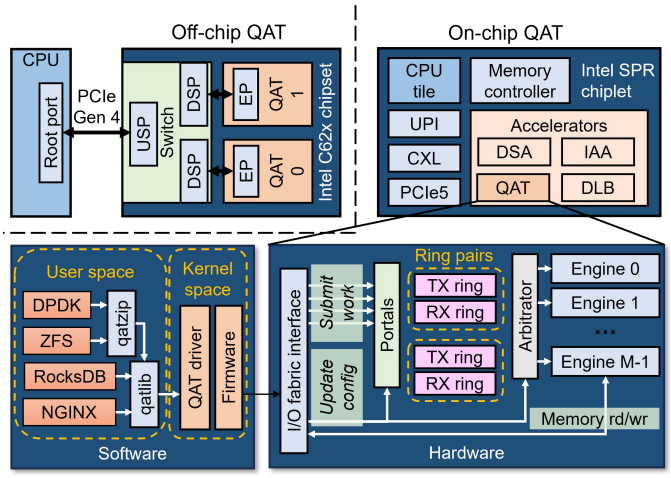


Fig. 1: Overview of the off-chip and on-chip QAT accelerator architectures and the software stack of the on-chip QAT accelerator.

SoC Features. Compared to off-chip QAT, on-chip QAT introduces new SoC-level features, including Shared Virtual Memory (SVM) and Scalable I/O virtualization (SIOV), supported by the latest QAT driver in Linux kernel 6.2 and above. SVM complies with the PCIe Address Translation Service (ATS) protocol, enabling QAT to access address translation information via IOMMU on the host side and the Address Translation Cache (ATC) on the device side. SVM enables QAT to directly access shared data via virtual addresses coherently, simplifying the deployment and avoiding the expensive memory pinning and data copying needed for DMA-based data transfer in off-chip QAT. Single Root I/O Virtualization (SR-IOV), introduced earlier, and SIOV both support sharing a single physical QAT as multiple virtual

Qirong Xia, Houxiang Ji, Yang Zhou, and Nam Sung Kim are with the University of Illinois Urbana-Champaign, Urbana, IL 61801 USA (e-mail: qirongx2@illinois.edu; hj14@illinois.edu; yangz15@illinois.edu; nskim@illinois.edu).

†Qirong Xia and Houxiang Ji contributed equally to this work.

This article has been accepted for publication in IEEE Computer Architecture Letters. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/LCA.2025.3534831

devices across multiple Virtual Machines (VMs) or containers. Nonetheless, SIOV offers greater scalability and flexibility than SR-IOV by moving the non-performance-critical parts of the device virtualization stack to software running on CPU and IOMMU whereas the virtualization stack of SR-IOV is limited to hardware. As such, SIOV can scale a single physical device to thousands of virtual devices instead of a fixed number of virtual devices in SR-IOV.

Software Interface. Multiple user-space libraries and plugins, `e.g. QATzip` [2], `QATlib` [3], are provided for seamless utilization of QAT in applications. These user-space libraries convert function calls to QAT from applications into descriptors, which are passed to the kernel-space QAT driver [4], parsed by the firmware, and finally submitted to QAT. The QAT driver, implemented as a Linux kernel module, also enables direct in-kernel usage of QAT with specific Linux kernel APIs [5].

Related Work. Intel In-Memory Analytics Accelerator (IAA) [6], another on-chip accelerator on SPR, offers compression functionality but only supports `Deflate` algorithm. The IBM Power7+ processor [7] includes an on-chip compression accelerator but relies on slower DMA for data transfer due to the lack of the cache-coherent interconnect and SVM support.

II. QAT Compression Capability Characterization

Using two benchmarks, we first evaluate the latency and throughput of QAT across different configurations to explore the efficient use of QAT usage. Additionally, we compare the latency and throughput of QAT with those of off-chip QAT accelerators, and the CPU to highlight the performance improvements across QAT generations and the compression acceleration offered by QAT over the CPU.

A. Methodology

System and Devices. Table I summarizes the specifications of our systems, as well as two QAT accelerators. The off-chip QAT accelerator is installed on the Intel 3rd-generation Scalable Processor, codenamed Ice Lake. While Ice Lake has 40 cores per CPU, we offline 8 of them to make a fair comparison with SPR which has only 32 cores. Additionally, we disable hyper-threading and lock the CPU frequency on both systems for measurement stability.

Benchmarks. To characterize the compression capability of QAT, we use HyperCompressBench [10], referred to as `HCB` in this work, and SPEC CPU 2017 [11]. `HCB` is a suite of synthetic compression workloads, created by integrating four standard benchmarks, Silesia, Canterbury, Calgary, and SnappyFiles, to reflect statistical characteristics of compression workloads in the Google datacenter. We use `ptrace` to periodically capture memory pages used by the SPEC benchmarks, following the same methodology as prior work [12], and use them as workloads, denoted as `SPEC-Mem` in this work.

TABLE I: System and devices.

SPR system	Description
OS (kernel)	Ubuntu 22.04.2 LTS (Linux kernel v6.2)
CPU	Intel Xeon Gold 6438Y+ CPU [8]@2.0GHz, 32 cores 60MB LLC per CPU, Hyper-Threading disabled
Memory	Socket 0: 8× DDR5-4800 channels
On-chip QAT	PCIe 5.0, 32GT/s per lane
Ice Lake system	**Description**
OS (kernel)	Ubuntu 22.04.2 LTS (Linux kernel v6.2)
CPU	Intel Xeon Platinum 8380 CPU [9]@2.0GHz, 40 cores 60MB LLC per CPU, Hyper-Threading disabled
Memory	Socket 0: 8× DDR5-4800 channels
Off-chip QAT	QAT C62x adapter, PCIe 4.0, 16GT/s per lane

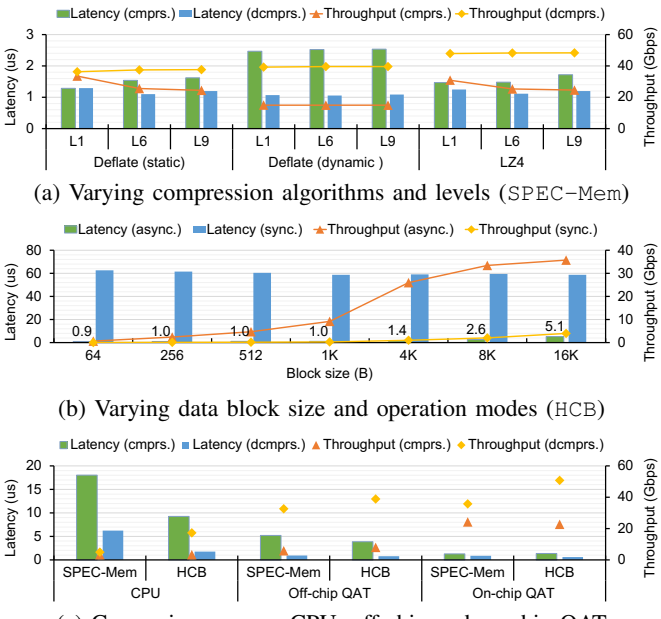
Metrics. QAT conducts (de)compression in a streaming way by dividing a given workload into N blocks and (de)compressing each block independently without any history of the previous (de)compressed blocks. We use the average compression latency per block (*i.e.*, the total compression latency divided by N), and the average compression throughput (*i.e.*, the size divided by the total compression latency) as the performance metrics. Note that a single CPU thread submits blocks to QAT when measuring latency, while all 32 threads are used to assess throughput.

B. Characterization

We leverage the Intel QAT driver [4] and configure QAT via `sysfs`, specifying the compression algorithms, compression levels, data block sizes, and operation modes. Each configuration is tested 10 times with workloads consisting of more than 100K blocks.

Compression algorithms and levels. We evaluate `Deflate` (with static and dynamic Huffman coding) and `LZ4` as representative compression algorithms provided by QAT. A higher compression level offers a higher degree or intensity of compression (*i.e.*, a higher compression ratio) but at the cost of a longer compression time.

`Deflate` with dynamic Huffman coding presents the highest latency, while lightweight `LZ4` achieves the lowest latency when compressing/decompressing 4 KB blocks from `SPEC-Mem`. For a given compression algorithm and block size, as the compression level increases, the compression ratio (*i.e.*,



(a) Varying compression algorithms and levels (`SPEC-Mem`)

(b) Varying data block size and operation modes (`HCB`)

(c) Comparison among CPU, off-chip and on-chip QAT

Fig. 2: Comparison of compression performance.

This article has been accepted for publication in IEEE Computer Architecture Letters. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/LCA.2025.3534831

compressed block size) decreases, but the latency increases, resulting in a decrease in throughput, as shown in Fig. 2a and Table II.

Insight 1: Lower compression level is preferred where latency is critical without notably compromising the compression ratio.

Block size and operation modes. As discussed in § II-A, QAT divides a given workload into blocks, we choose block sizes ranging from 64B to 16KB for characterization. QAT supports two operation modes: synchronous and asynchronous. In synchronous mode, the host CPU submits a compression task and waits for its completion before starting to work on the next block. In asynchronous mode, the host CPU submits all tasks at once and context-switches to another process. When QAT finishes the tasks, it sends an interrupt to notify the compression process of the completion.

Fig. 2b shows the latency and throughput for various block sizes in two operation modes running on `HCB`, using `Deflate` with static Huffman coding at compression level 1. The asynchronous mode achieves 7.7–38.6× lower latency than the synchronous mode across these block sizes as it can process multiple compression tasks in parallel. As the block size increases, it takes higher latency for QAT to retrieve and process data but the throughput can also increase as more data are compressed and thus the data movement overhead is amortized. The trends in latency and throughput are consistent with those seen on `SPEC-Mem`.

Insight 2: Larger data blocks and asynchronous operations significantly boost QAT's compression performance across various scenarios.

Comparison with off-chip QAT and CPU. We compare the compression performance of CPU, off-chip QAT, and on-chip QAT, all of which run `Deflate` with static Huffman coding, compression level 6, and 4KB data block size (Fig. 2c). The compression ratios obtained by QAT match those of the software versions of the algorithms on CPU. CPU-based `Deflate` presents 17.7× and 6.1× longer compression and decompression latency than on-chip QAT, respectively. The performance improvement of on-chip QAT over older off-chip QAT is attributed to the on-chip integration minimizing communication delay and leveraging the higher bandwidth of PCIe 5.0 for faster data transfers.

Insight 3: On-chip QAT consistently delivers lower latency and higher throughput than both the CPU and off-chip QAT, showcasing its superior compression capability and efficiency for demanding workloads.

TABLE II: Compression ratios of various compression algorithms and levels

Benchmark	Algorithm	Level 1	Level 6	Level 9
`HCB`	Deflate (dynamic)	0.28	0.27	0.27
	Deflate (static)	0.32	0.31	0.31
	LZ4	0.38	0.36	0.36
`SPEC-Mem`	Deflate (static)	0.56	0.55	0.55
	Deflate (dynamic)	0.50	0.49	0.49
	LZ4	0.64	0.63	0.63

III. QAT-based Kernel-Space Memory Compression Acceleration

In this section, we reimplement a kernel feature, `zswap`, as a case study to showcase how QAT's compression capability enhances system performance. By offloading computation-intensive compression to QAT, QAT-`zswap` reduces disruption to co-running applications notably compared to CPU-`zswap`. While this section highlights `zswap` 's potential with QAT, its benefits can extend to a broader range of compression workloads in datacenters.

A. QAT-based `zswap`

`zswap` [13] is a Linux kernel feature that compresses swap-out pages and stores the compressed pages in a memory pool (*i.e.*, `zpool`) instead of writing them back to backing storage directly, thereby reducing disk I/O and enabling faster page fault handling when the page fault hits on the zpool. `zswap` is triggered when the number of free memory pages drops below a certain threshold. Lines with orange markers Fig. 3 depict a page compression in `zswap` running on the host CPU (`cpu-zswap`). When `zswap` swaps out a page, the host CPU ① receives the page address and destination address in `zpool`, ② reads the page from memory, polluting the LLC with data in the page, ③ compresses the page using the CPU cores, consuming a notable number of CPU cycles, ④ stores the compressed page into `zpool` and ⑤ returns compressed size, and informs `zswap` of the completion for further processing. The side effects of ② and ③ exacerbate contention of CPU cycles and cache space between `zswap` and co-running applications, hurting application performance during `zswap`ing [14].

To alleviate the aforementioned contention while preserving the benefits of `zswap`, we offload the data- and compute-intensive operations of compression/decompression to QAT (`qat-zswap`) instead of running it on the host CPU. For both `cpu-zswap` and `qat-zswap`, we choose `Deflate` that is most commonly used although they support various compression algorithms, including `LZ4` and `LZO`. We first register the QAT-accelerated `Deflate` function (`qat_deflate`) to the Linux Crypto API and expose it to `zswap` in the kernel space. When there is a page to compress, `qat-zswap` calls the `qat_deflate` instead of the CPU-based `Deflate` function. `qat_deflate` call is converted to a work descriptor by the

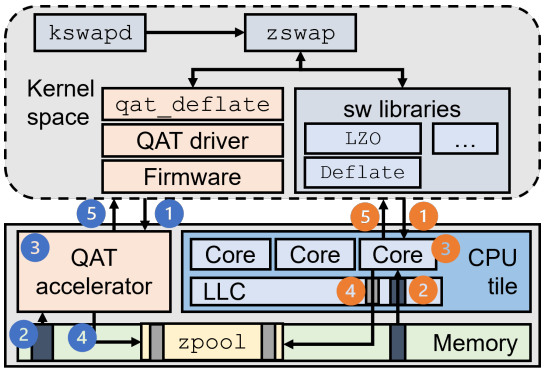


Fig. 3: `zswap` running on CPU (orange) and QAT (blue).

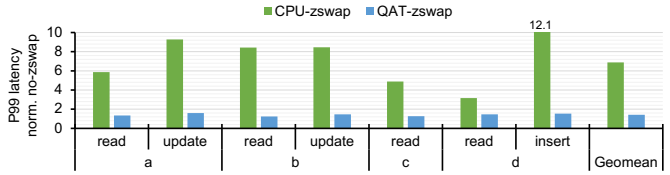


Fig. 4: 99th-percentile (p99) latency of `Redis` with `YCSB` workloads running on `cpu-zswap` and `qat-zswap`, normalized to no `zswap` running.

QAT driver and firmware, and then submits this descriptor to QAT. After receiving the descriptor, `qat-zswap` follows the same steps as `cpu-zswap` depicted by lines with blue markers Fig. 3. As QAT is now responsible for all data movement and compression, `qat-zswap` no longer causes cache pollution and frees the CPU cores from compute-intensive compression.

B. Evaluation

Application. We use `Redis` [15] and `YCSB` [16] to evaluate the impact of `cpu-zswap` and `qat-zswap` on co-running latency-sensitive applications. `Redis` is a popular in-memory Key-Value Store (KVS) database, and `YCSB` is used as workloads for `Redis`. We use four YCSB workloads: (a) update heavy (50% read and 50% write), (b) read heavy (95% read and 5% write), (c) read-only (100% read), and (d) read latest (95% read and 5% insert) with a `zipfian` distribution.

Methodology. We co-run `Redis` and a SPEC CPU 2017 benchmark on SPR (§I). More specifically, we co-run 16-threaded `631.deepsjeng_s` with a total memory limit of 10GB when launching the `YCSB` workload on `Redis`. The SPEC benchmark serves as the memory stressor to trigger `zswap` and we ensure only pages consumed by the benchmark are swapped out by `cgroup`. We assign 14 `Redis` clients on 14 different CPU cores to submit the workloads to 1 `Redis` server. `zswap` is assigned to the same CPU core as the `Redis` server.

Tail latency. We use the 99th-percentile (p99) latency as the performance metric for `Redis`. Fig. 4 plots the p99 latency of `Redis` running with `cpu-zswap` and `qat-zswap`, normalized to those of `Redis` running alone (`no-zswap`). `cpu-zswap` increases the p99 latency by 3.2–12.1× compared to `no-zswap`. `qat-zswap` reduces the increase to 1.2–1.6× as QAT executes the heavy compression replacing CPU.

CPU cycle consumption and cache miss ratio. Table III shows a 9–15% CPU utilization decreases from `cpu-zswap` to `qat-zswap` as the CPU is only used for submitting and receiving the QAT requests if the (de)compression is offloaded to the QAT, rather than performing the full (de)compression algorithms. The cache miss rate in `qat-zswap` is reduced by 3%-13%, compared to `cpu-zswap`. The QAT accelerator's usage of DMA allows reading from and writing to the main memory directly without touching the CPU's cache hierarchy.

TABLE III: CPU utilization and cache misses of `cpu-zswap` and `qat-zswap`

	Workload	CPU-zswap	QAT-zswap
CPU utilization	YCSB-a	19%	4%
	YCSB-b	14%	5%
	YCSB-c	15%	4%
	YCSB-d	15%	4%
Cache miss rate	YCSB-a	30%	16%
	YCSB-b	24%	20%
	YCSB-c	27%	24%
	YCSB-d	23%	19%

Insight 4: Offloading compression workloads to QAT can significantly save CPU resource consumption and thus boost overall system performance.

IV. Conclusion

This work explores the compression capabilities of the on-chip QAT accelerator integrated into the Intel 4th generation SPR CPU. We characterize the on-chip QAT accelerator regarding latency and throughput under different compression algorithms, compression levels, and operation modes. Based on the characterization, we offload compression in `zswap` to the QAT accelerator and achieve 2.2–7.9× tail latency reduction on co-running applications compared to `zswap` running on CPU.

References

[1] Intel Corporation, "Intel quickassist adapter family for servers," https://www.intel.com/content/www/us/en/products/docs/network-io/ethernet/10-25-40-gigabit-adapters/quickassist-adapter-for-servers.html, accessed in 2024.
[2] Intel Corporation, "Intel QuickAssist Technology (QAT) QATzip Library," https://github.com/intel/QATzip, accessed in 2024.
[3] Intel Corporation, "Intel QuickAssist Technology Library (QATlib)," https://github.com/intel/qatlib, accessed in 2024.
[4] Intel Corporation, "Intel QuickAssist Technology Driver for Linux – HW Version 2.0," https://www.intel.com/content/www/us/en/download/765501/intel-quickassist-technology-driver-for-linux-hw-version-2-0.html, accessed in 2024.
[5] Linux Kernel Documentation Team, "Linux kernel crypto api," https://www.kernel.org/doc/html/v6.2/crypto/, accessed in 2024.
[6] Intel Corporation, "Intel® in-memory analytics accelerator (intel® iaa)," https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/in-memory-analytics-accelerator.html, accessed in 2024.
[7] B. Blaner, B. Abali, B. M. Bass, S. Chari, R. Kalla, S. Kunkel, K. Lauricella, R. Leavens, J. J. Reilly, and P. A. Sandon, "Ibm power7+ processor on-chip accelerators for cryptography and active memory expansion," *IBM Journal of Research and Development*, 2013.
[8] Intel Corporation, "Intel® Xeon® Gold 6438Y+ Processor," https://ark.intel.com/content/www/us/en/ark/products/232382/intel-xeon-gold-6438y-processor-60m-cache-2-00-ghz.html, accessed in 2024.
[9] Intel Corporation, "Intel® Xeon® Platinum 8380 Processor," https://www.intel.com/content/www/us/en/products/sku/212287/intel-xeon-platinum-8380-processor-60m-cache-2-30-ghz/specifications.html, accessed in 2024.
[10] Google, "Hypercompressbench," https://github.com/google/HyperCompressBench?tab=readme-ov-file, accessed in 2024.
[11] Standard Performance Evaluation Corporation, "SPEC CPU 2017 Benchmark," https://www.spec.org/cpu2017/, accessed in 2024.
[12] S. Sardashti and D. A. Wood, "Could compression be of general use? evaluating memory compression across domains," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2017.
[13] Linux Kernel Documentation Team, "Linux kernel documentation: zswap," https://www.kernel.org/doc/html/v6.0/admin-guide/mm/zswap.html, accessed in 2024.
[14] H. Ji, M. Mansi, Y. Sun, Y. Yuan, J. Huang, R. Kuper, M. M. Swift, and N. S. Kim, "STYX: Exploiting SmartNIC capability to reduce datacenter memory tax," in *Proceedings of the 2023 USENIX Annual Technical Conference (USENIX ATC'23)*, 2023.
[15] Redis Labs, "Redis: The Real-Time Data Platform," https://redis.io, accessed in 2024.
[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*, 2010.