CDPU: Co-designing Compression and Decompression Processing Units for Hyperscale Systems

Sagar Karandikar
UC Berkeley, Google
Berkeley, CA, USA

Aniruddha N. Udipi
Google
Mountain View, CA, USA

Junsun Choi
UC Berkeley
Berkeley, CA, USA

Joonho Whangbo
UC Berkeley
Berkeley, CA, USA

Jerry Zhao
UC Berkeley
Berkeley, CA, USA

Svilen Kanev
Google
Mountain View, CA, USA

Edwin Lim
UC Berkeley
Berkeley, CA, USA

Jyrki Alakuijala
Google
Zürich, Switzerland

Vrishab Madduri
UC Berkeley
Berkeley, CA, USA

Yakun Sophia Shao
UC Berkeley
Berkeley, CA, USA

Borivoje Nikolić
UC Berkeley
Berkeley, CA, USA

Krste Asanović
UC Berkeley
Berkeley, CA, USA

Parthasarathy Ranganathan
Google
Mountain View, CA, USA

ABSTRACT

General-purpose lossless data compression and decompression ("(de)compression") are used widely in hyperscale systems and are key "datacenter taxes". However, designing optimal hardware compression and decompression processing units ("CDPUs") is challenging due to the variety of algorithms deployed, input data characteristics, and evolving costs of CPU cycles, network bandwidth, and memory/storage capacities.

To navigate this vast design space, we present the first large-scale data-driven analysis of (de)compression usage at a major cloud provider by profiling Google's datacenter fleet. We find that (de)compression consumes 2.9% of fleet CPU cycles and 10-50% of cycles in key services. Demand is also artificially limited; 95% of bytes compressed in the fleet use less capable algorithms to reduce compute, motivating a CDPU that changes cost vs. size tradeoffs.

Prior work has improved the microarchitectural state-of-the-art for CDPUs supporting various algorithms in fixed contexts. However, we find that higher-level design parameters like CDPU placement, hash table sizing, history window sizes, and more have as significant of an impact on the viability of CDPU integration, but are not well-studied. Thus, we present the first end-to-end design/evaluation framework for CDPUs, including: 1. An open-source RTL-based CDPU generator that supports many run-time and compile-time parameters. 2. Integration into an open-source RISC-V SoC for rapid performance and silicon area evaluation across CDPU placements and parameters. 3. An open-source (de)compression benchmark, HyperCompressBench, that is representative of (de)compression usage in Google's fleet.

Using our framework, we perform an extensive design space exploration running HyperCompressBench. Our exploration spans a 46× range in CDPU speedup, 3× range in silicon area (for a single pipeline), and evaluates a variety of CDPU integration techniques to optimize CDPU designs for hyperscale contexts. Our final hyperscale-optimized CDPU instances are up to 10× to 16× faster than a single Xeon core, while consuming a small fraction (as little as 2.4% to 4.7%) of the area.

CCS CONCEPTS

• **Computer systems organization → Architectures**; **Cloud computing**; • **Hardware → Communication hardware, interfaces and storage**; **Application-specific VLSI designs**; • **Information systems → Data compression**.

KEYWORDS

compression, decompression, hardware-acceleration, warehouse-scale computing, hyperscale systems, profiling

ACM Reference Format:
Sagar Karandikar, Aniruddha N. Udipi, Junsun Choi, Joonho Whangbo, Jerry Zhao, Svilen Kanev, Edwin Lim, Jyrki Alakuijala, Vrishab Madduri, Yakun Sophia Shao, Borivoje Nikolić, Krste Asanović, and Parthasarathy Ranganathan. 2023. CDPU: Co-designing Compression and Decompression Processing Units for Hyperscale Systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3579371.3589074

ISCA '23, June 17–21, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0095-8/23/06.
https://doi.org/10.1145/3579371.3589074

1 INTRODUCTION

As demand for cloud computing grows and traditional hardware scaling techniques slow down, improving the performance and efficiency of warehouse-scale computers (WSCs) through hardware-software co-design is increasingly critical. Complicating this, WSCs run large, layered software stacks consisting of diverse and rapidly evolving microservices, making specialization difficult. However, prior work has shown that several common *datacenter taxes* [41, 56] appear across hyperscale services and thus present specialization opportunities. Characterization and acceleration of several of these taxes has been explored in prior work [39, 42, 43, 49], but little attention has been paid to general-purpose lossless data compression and decompression (referred to as "(de)compression" in this work) in hyperscale contexts.

Most datacenter taxes implement critical *functionality* like inter-service communication, security, or memory movement. In contrast, (de)compression is unique in that its purpose is not to add functionality, but to *enable a trade-off* between the consumption of two classes of WSC resources: runtime (CPU cycles) and storage/communication capacity (persistent storage capacity, memory capacity [45, 61], and network bandwidth). Unlike other datacenter taxes, service developers must first decide whether to compress at all, and then select an algorithm that achieves satisfactory compression quality within their constraints.

This presents an interesting opportunity for hardware acceleration: an accelerator that radically outperforms software implementations can not only reduce existing CPU cycles in the fleet, but also increase compression usage in general, leading to additive savings in storage, memory, and network bandwidth. However, introducing specialized hardware complicates the design space; the total cost of ownership (TCO) calculation must now account for hardware complexity, area vs. performance, and more.

In this work, we present the first large-scale data-driven analysis of lossless data (de)compression usage at a major cloud provider by profiling Google's datacenter fleet. We find that (de)compression consumes 2.9% of fleet CPU cycles and 10% to 50% of CPU cycles in key services at Google. This demand is also artificially limited; 95% of bytes compressed in Google's fleet forgo more aggressive forms of compression because of the high compute cost, motivating HW acceleration that changes time vs. data size trade-offs. While profiling fleet usage is helpful, we also find that true co-design for (de)compression processing units (CDPUs) requires a comprehensive evaluation environment, due to the large number of high-level design parameters and their impact on end-to-end performance.

A large body of prior work has improved the microarchitectural state-of-the-art for CDPUs supporting various algorithms in fixed contexts [10–18, 29, 30, 32, 36, 46, 50, 53, 58, 59]. While these improvements are important, we find that higher-level design parameters like accelerator placement, hash table sizing, history window sizes, and more can have just as significant of an impact on the value and feasibility of CDPU integration, but are not well-studied in the literature. Thus, we present the first end-to-end design and evaluation framework for CDPUs, which includes: 1. An RTL-based CDPU generator that supports many run-time and compile-time configurable parameters. 2. Integration into a RISC-V SoC for rapid performance and silicon area evaluation with varying CDPU placements and configurations. 3. A (de)compression benchmark, HyperCompressBench, that is representative of (de)compression usage in Google's fleet. All components of this framework are open-source[1], enabling the community to build and evaluate CDPUs for both hyperscale systems and their own use cases.

Using our CDPU design framework, we perform an extensive design space exploration running HyperCompressBench. Our exploration spans a 46× range in accelerator speedup, 3× range in silicon area (for a single pipeline), and evaluates a variety of accelerator integration techniques to better understand optimal CDPU designs for hyperscale contexts. Our final hyperscale-optimized accelerator instances are up to 10× to 16× faster than a single Xeon core, while consuming a small fraction (as little as 2.4% to 4.7%) of the area.

2 COMPRESSION BACKGROUND

Compression algorithms are used to produce a reduced-size representation of source data that can later be fed to a decompressor to exactly reproduce the original data. While the functional goal is only to minimize the output size (maximizing the *compression ratio*, equal to uncompressed divided by compressed size), algorithms must also account for metrics like latency, throughput, and CPU/memory consumption, resulting in a vast design space. In a hyperscale context, compression reduces the consumption of several resources, including storage (bytes written to disk/SSD), network bandwidth (e.g., RPC traffic), and memory (transparently [45, 61] or via application managed compression). Compression can also implicitly save other resources such as caches, network-on-chip capacity, etc., but we do not explore these in this work.

2.1 Compression algorithm fundamentals

Compression algorithms generally contain two main components: a dictionary-coding stage and an entropy-coding stage. During *dictionary coding*, data size is reduced by searching for matches between the input data and a "dictionary" of known values, then encoding the input in terms of the "best" match in the dictionary, deduplicating repeated strings in the input. LZ77 [65] is a widely-used dictionary coding algorithm that uses a sliding window of already processed input data as the dictionary. Matches are encoded as triplets of `(offset, length, literal)`. Such a triplet indicates to the decompressor that `length` bytes should be copied to the output starting from `offset` bytes back in the window of output generated so far. Then, the raw `literal` is also copied to the output, for example to encode data when no matches were found in the dictionary.

Entropy coding compresses symbols (e.g. `(offset, length, literal)` triplets produced by LZ77) by representing more commonly occurring symbols with fewer bits. Popular techniques include Huffman coding [37], arithmetic coding, and Asymmetric Numerical Systems (ANS) [2, 35]. Huffman and arithmetic coding

[1] **CDPU generator, custom Chipyard, custom FireSim:**
https://github.com/ucb-bar/compress-acc
HyperCompressBench:
https://github.com/google/HyperCompressBench
Archival URLs:
See Artifact Appendix (Appendix A)

trade-off compression ratio and performance—Huffman is cheaper in CPU cycle cost, but arithmetic coding generally achieves a better compression ratio. ANS (such as tANS/FSE [2, 35]) combines the best of both worlds, with low CPU cost and high compression ratio.

2.2 Compression algorithm taxonomy

Compression algorithm developers trade-off compression ratio vs. performance by combining these components in novel ways and tuning parameters within them. For example, they can choose how much effort to expend trying to find an "optimal" match during LZ77-style dictionary coding or change the size of the sliding history window. A larger *window size* typically yields better compression ratios, but must be bounded to limit memory consumption. Many algorithms accept a *compression-level* parameter, which also allows users to tune algorithm performance.

In Section 3, we will analyze six algorithms that are used in Google's fleet. We qualitatively group these into "heavyweight" and "lightweight" classes (which we will justify quantitatively in Section 3.3):

Heavyweight algorithms: These prioritize compression ratio over speed. They generally have a large space of parameters and use sophisticated LZ77/entropy-coding techniques.

- **ZStd** [8, 31]: LZ77/Huff./FSE. Params: comp. level + window size.
- **Flate** [7, 34]: LZ77/Huff. Params: comp. level + window size.
- **Brotli** [1, 20]: LZ77/Huff./context modeling/static dictionary [19]. Params: compression level + window size.

Lightweight algorithms: These prioritize speed over compression ratio. They generally use "LZ77-inspired" dictionary coding, little or no entropy coding, and have few parameters.

- **Snappy** [5, 9]: LZ77-inspired, no entropy coding. Fixed window size (64 KiB), no compression levels.
- **Gipfeli** [3, 47]: LZ77-inspired, simple entropy coding. Fixed window size (64 KiB), no compression levels.
- **LZO** [4, 57]: LZ77-inspired dictionary coding, no entropy coding. Supports compression levels.

ZStd and Brotli can also become more lightweight by setting a low compression level. We will explore this in Section 3.3.3.

3 PROFILING COMPRESSION USAGE AT HYPERSCALE

In this section, we profile the fleet-wide usage of (de)compression in Google's datacenters to motivate the design of a CDPU and understand design constraints.

3.1 Data Sources

3.1.1 Fleet-wide CPU Cycle Data. Google's infrastructure provides fleet-wide runtime information about CPU-cycle consumption using a sampling framework, Google-Wide Profiling (GWP) [52], that randomly samples fleet servers. When a server is profiled, the sampler collects profiles including workload names, stack traces, and cycle counts, enabling determination of where time is spent in the software stack. We use this to classify fleet-wide CPU-cycles spent in (de)compression by algorithm.

3.1.2 Fleet-wide compression/decompression call sampling. An extension of this sampling framework also enables detailed profiling of (de)compression calls in userspace, including collecting the algorithm used, input and output sizes, window sizes, and compression levels. Given the additional engineering effort this requires, data is only collected for the Snappy, ZStd, Flate, and Brotli algorithms, which, as Figure 1 shows, are the dominant algorithms in the fleet.

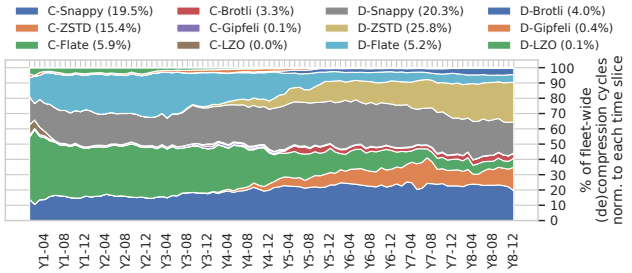


Figure 1: Percentage of (de)compression cycles in Google's fleet over several years, broken down by algorithm and normalized to each month. C=compress, D=decompress.

3.2 Opportunity for (De)compression Acceleration

WSCs today spend significant compute on (de)compression. In Google's infrastructure, 2.9%[2] of *fleet-wide* CPU cycles are spent in (de)compression; 56% of these cycles are spent in decompression and the rest in compression.

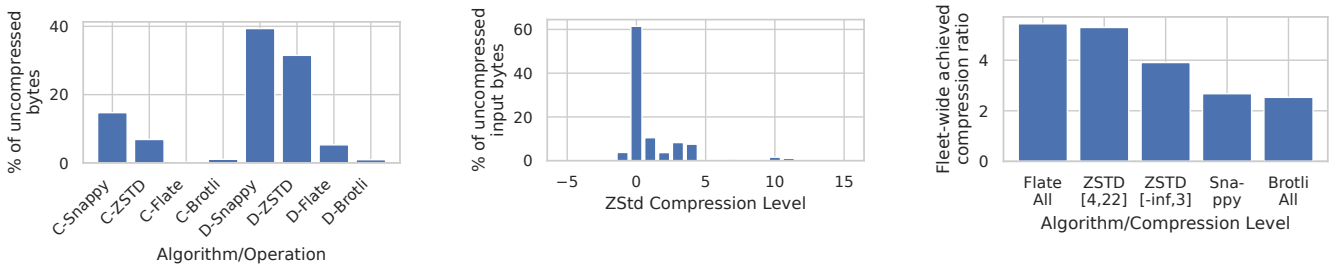
For large services, (de)compression can be a much greater proportion of total cycle consumption. We find that a total of sixteen services constitute around half of all fleet-wide cycles for Snappy and ZStd[3] (de)compression. Out of these, one service spends nearly 50% of its total cycles on (de)compression, another spends over 35%, and eight more spend between 10% and 25% of their cycles each on (de)compression. *Even ignoring potential growth in demand, these represent a significant acceleration opportunity at hyperscale.*

3.3 Can accelerators change WSC resource tradeoffs?

While reducing the existing CPU cycles spent on (de)compression is a useful goal, it is important to note that this usage is a function of the performance constraints imposed by current software libraries. When considering the introduction of specialized hardware, we must keep in mind that the accelerator is likely to change the "space" (storage/memory bytes, network bandwidth) vs. "time" (runtime, CPU cycles) trade-off involved in selecting a compression algorithm, that algorithm's parameters, or indeed, choosing to compress at all in a given situation.

In an ideal scenario, the accelerator would sufficiently reduce the performance overhead of "heavyweight" forms of (de)compression such that services can always choose them over "lightweight" techniques (or even no compression), and reduce storage, memory, and network bandwidth consumption for "free". To understand this opportunity, we must answer four key questions:

[2] At hyperscale, this can translate to 100s of millions of dollars [24, 56].
[3] In the rest of the paper, we focus on Snappy and ZStd as dominant representatives of "lightweight" and "heavyweight" algorithms in the fleet respectively.



(a) Fleet-wide uncompressed bytes handled by (de)compression, broken down by algo.

(b) Fleet-wide ZStd compression level distribution.

(c) Aggregate fleet-wide compression ratios achieved, by algo./compression level pairs.

Figure 2: Google fleet-wide (de)compression algorithm breakdowns. C=compression, D=decompression.

3.3.1 Do existing services prefer to use heavyweight or lightweight algorithms? Figure 1 shows a detailed breakdown of CPU time spent in the fleet on compression and decompression by algorithm, self-normalized to each month. In this sub-section, we focus only on the final time slice, which is summarized in the legend. In addition to cycle consumption, we would also like to understand the *amount of data* that each algorithm is invoked on. Figure 2a thus differentiates algorithms based on the number of uncompressed bytes they handle in the fleet (i.e., compression inputs and decompression outputs).

We find several interesting trends from this data. For compression, where the heavyweight vs. lightweight distinction is most significant, we see that slightly more *cycles*, 56%, are spent in heavyweight compression. However, from the perspective of *bytes* handled, the outcome is *reversed*: heavyweight compression only accounts for 36% of the total. This foreshadows the difference in cost-per-byte-compressed between heavyweight and lightweight compression, explored in greater detail in Section 3.3.4. In decompression, the CPU consumption imbalance between heavyweight and lightweight is far more stark, but the cost-per-byte is closer: heavyweight algorithms comprise 63% of fleet decompression *cycles*, while producing 49% of uncompressed *bytes*.

As an aside, Figure 2a also shows an interesting insight: on average, each byte that is compressed in the fleet is decompressed 3.3 times. So, despite a lower cost-per-byte, decompression remains a worthy target for hardware acceleration. Further, decompression is often more performance-sensitive, naturally appearing on client-visible read paths, rather than usually non-critical write paths.

3.3.2 Are heavyweight algorithms used to their full potential? Generally, this requires supplying a larger *compression-level* argument to the algorithm, instructing it to spend more cycles improving the compression ratio. Consider ZStd compression, which currently supports levels from negative infinity to 22. Figure 2b shows the distribution of bytes passed to ZStd compression calls in the fleet, binned by the associated compression level specified by the caller. We find that even services that use ZStd tend to avoid high compression levels: 88% of bytes are compressed at level 3 (the default) or lower, while over 95% of bytes are compressed at level 5 or lower. Fewer than 0.002% of bytes are compressed at levels $\geq$12.

Combining the data in Figures 2a and 2b we glean a critical insight: over 95% of bytes compressed in the fleet are handled either by a lightweight algorithm (Snappy) or a heavyweight algorithm at low compression level (ZStd at level $\leq$ 3). This suggests that *there is significant opportunity for an accelerator that can achieve higher compression ratios within existing performance bounds* to produce significant savings in storage, network, and memory consumption.

3.3.3 Do high compression levels result in improved compression ratio? Of course, the goal of using heavyweight algorithms configured to high compression levels is to achieve a better compression ratio. Therefore, we must understand whether this improvement is indeed notable.

Before we present this data, it is important to caveat that extrapolating from this data is generally difficult due to the highly data-dependent nature of both compression ratio and cycles-per-byte terms. While the data gives the reader an estimate of possible improvements and is valuable because it is based on large fleet byte volumes, a true comparison of algorithms/levels requires running the same sets of representative data through algorithms/levels of interest. We will address this in Section 4 when we construct our benchmark suites.

Figure 2c shows the aggregate fleet-wide compression ratio achieved by each compression algorithm (*total* uncompressed bytes divided by *total* compressed bytes). ZStd bins are further separated by the user-specified compression level. We can see that compression is clearly beneficial across the fleet, with no algorithm having an aggregate compression ratio less than 2. Furthermore, the data aligns with expectations from the taxonomy we established in Section 2.2. ZStd and Flate clearly belong in the heavyweight category, exceeding Snappy's compression ratio even at the lowest compression levels. Brotli results do not align with our taxonomy because most of its usage in the fleet is at low compression levels.

Quantitatively, we observe a favorable trend in the data to justify hardware acceleration. Services that use ZStd at a low compression level achieve a 1.46× improved compression ratio over services that use Snappy. Services that use ZStd at a high compression level achieve an additional 1.35× improved compression ratio over services that use it at a low compression level. It is also important to note that this is likely under-representing the potential of ZStd's highest compression-levels, since Figure 2b showed that the vast majority of bytes in the `[4, 22]` bin in Figure 2c are compressed at level 4, due to the aforementioned performance constraints.

In a hyperscale context, the corresponding reductions in demand for storage, network, and memory capacity that arise from these

differences in compression ratio can translate to a further potential savings of hundreds of millions of dollars across the industry [24, 43, 56], in addition to the savings from CPU-cycle reduction/offloading: most hyperscaler customers are big-data companies who spend as much on storage as compute [51], while memory has been shown to be 50% of WSC TCO [26], and providing sufficient network bandwidth at low cost is a perpetual concern for hyperscalers [54].

3.3.4 What is the cycle cost in software of using heavyweight algorithms at high compression level in the fleet? Is hardware acceleration necessary? Given the marked difference in achieved compression ratio using different algorithms/levels, one might ask: why not simply migrate to heavyweight algorithms at high compression levels in software?

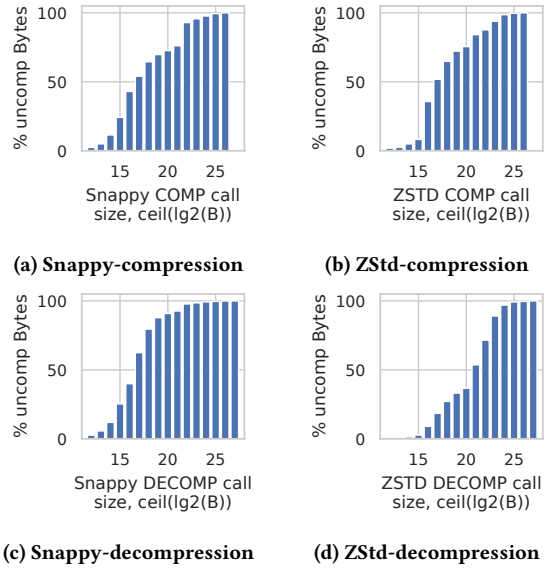
To answer this question, we collect data on the aggregate cost-per-byte observed in the fleet for each algorithm, operation, and compression level of interest thus far. We elide the plot due to space constraints, however we find that our taxonomy from Section 2.2 is largely validated: both heavyweight compression and decompression are more expensive per-byte than lightweight compression and decompression respectively. We also find that services that use ZStd compression at lower compression levels pay 1.55× the cost-per-byte for compression as compared to those that use Snappy, and services that use ZStd compression at higher compression levels over lower compression levels pay an additional 2.39× cost-per-byte.

Extrapolating from this data (and keeping in mind caveats about the data-dependent nature of compression), if a service spends 25% of its cycles on Snappy compression (e.g., the services described in Section 3.2), switching to the highest ZStd levels would result in a 67% increase in the service's cycle consumption, a non-starter. There is also a significant additional cost for decompression, when the data is accessed later; ZStd decompression is 1.63× more costly than Snappy decompression, partially due to the entropy decoding.

Altogether, this profiling data suggests that there is significant headroom for services to achieve improved compression ratios for the deployed algorithms, but the cost of these algorithms in software is too high for services to adopt them. This suggests that *hardware-accelerated compression has the opportunity to save not only CPU cycles, but also to save storage/memory/network resources by changing the trade-off space between performance and compression ratio.*

3.4 Algorithm evolution vs. hardware accelerator design cycles

Even when hardware acceleration is well-motivated by projected resource savings, a significant roadblock to adoption is the opportunity cost of "ossification" of logic in hardware, since hardware design cycles are significantly longer than software development cycles. However, given the need for long-term stability of compression algorithms (e.g. for data written to cold storage), significant algorithm change generally only occurs when a completely new algorithm is adopted by a service. Referring back to Figure 1, we can observe the introduction of the ZStd algorithm in Google's fleet, which took roughly a year from being introduced to consuming 10% of fleet (de)compression cycles. While this broadly aligns



(a) Snappy-compression **(b) ZStd-compression**

(c) Snappy-decompression **(d) ZStd-decompression**

Figure 3: Cumulative call size distributions for Snappy/ZStd (de)compression. The x-axis bins calls by $log2(callsize)$, using uncompressed sizes. The y-axis is weighted by call size.

with hardware design cycles, starting a design from scratch and deploying it in this timeframe would be challenging.

This suggests that *an agile hardware development approach is necessary*, with early hardware/software co-design with algorithm developers and utilization of (de)compression accelerator *generators* that provide high-performance primitives that are common across multiple algorithms, alleviating the need to write entire accelerators from scratch. For example, transitioning from Flate to ZStd would mostly entail adding an FSE module. This methodology is explored in Section 5.

When compared to other datacenter taxes, (de)compression also has a qualitative advantage when considering hardware acceleration feasibility: the user API for compression and decompression has been essentially unchanged since the first compression tools were created—a stateless, buffer-in, buffer-out API, sometimes with a separate dictionary, and a streaming equivalent.

3.5 (De)Compression Accelerator Placement

In this section, we discuss the factors impacting an important choice in CDPU design: where to place it in the system: on-die, on a PCIe-attached device, or on a chiplet.

3.5.1 (De)compression call granularity. The granularity of offloaded work—in this case, the number of bytes to be (de)compressed—is a key factor in determining placement, since any overhead per accelerator invocation is only amortized over each payload size.

Figures 3a and 3b show the cumulative distribution of call granularities for Snappy and ZStd compression. Snappy's distribution is slightly more biased towards smaller calls: 24% of bytes compressed are from calls of size 32 KiB or smaller; for ZStd only 8% fall in this group. Interestingly, the distributions align near the median, with the 50th percentile of uncompressed bytes falling between 64 and

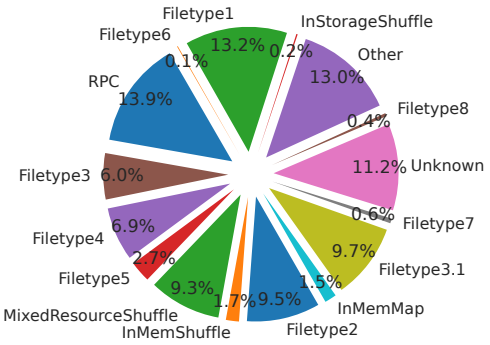
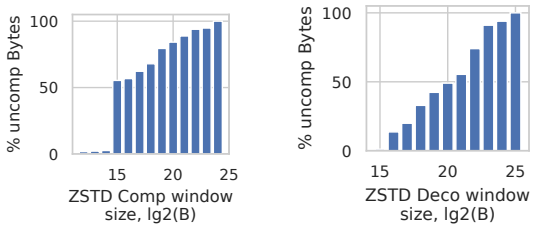


Figure 4: Percent of Google fleet-wide (de)compression cycles by the library that led to the (de)compression call.



(a) ZStd-compression **(b) ZStd-decompression**

Figure 5: Window size distributions for ZStd (de)compression in Google's fleet. The x-axis bins calls by $log2(windowsize)$. The y-axis is weighted by call size.

128 KiB calls in both. For ZStd, much of this jump comes from the `(32 KiB, 64 KiB]` bin, which represents 28% of bytes compressed. Apart from the 16.8% of bytes compressed by Snappy in the `(2 MiB, 4 MiB)` bin, both distributions increase uniformly until reaching a maximum size of 64 MiB.

Figures 3c and 3d show the corresponding data for decompression. Immediately, we see that Snappy's decompression distribution is slightly more biased towards smaller calls than its compression distribution, with 62% of bytes handled in calls smaller than 128 KiB and 80% of bytes handled in calls smaller than 256 KiB. On the other hand, the ZStd decompression distribution shifts drastically towards larger sizes, with the median size between 1 MiB and 2 MiB, rather than between 64 KiB and 128 KiB as for compression.

A back of the envelope projection of accelerator performance ranges shows that these distributions are insufficiently skewed to immediately fix accelerator placement. In contrast, if hypothetically most calls were 32 MB, a PCIe-attached accelerator would be a natural choice. As we will see in Section 6, both call sizes and various accelerator tuning parameters play important roles in determining accelerator placement; a comprehensive design-space exploration will be necessary to make a final determination.

3.5.2 Interaction with Related Accelerators. With increasing hardware specialization, we envision a future where our (de)compression accelerator is invoked in conjunction with related accelerators (e.g., a hardware protocol buffer (de)serializer [39, 43, 49]) as part of a larger data-access operation. While the hardware benefits of such a system are self-evident, the corresponding software services and libraries need to be architected appropriately as well.

Figure 4 shows fleet (de)compression cycles classified by the codebase (e.g. a library) that directly called the (de)compression operation. Note that 49% of cycles are derived from "file formats". Upon closer examination, we notice that even if these formats are internally "serializing and compressing protobufs" before writing to file, there are often small, unrelated book-keeping operations between the two accelerated operations. Services may also expect to pass in a sequence of serialized protobufs that are accumulated and compressed periodically. Handling these in hardware introduces significant complexity due to file-format specific logic and the need to track outstanding state, and can also limit file-format evolution.

This argues for *placing both accelerators close to the CPU cores*, utilizing the CPU caches or even main memory as the intermediate storage, allowing the general-purpose cores to sequence data movement between them in the normal course of program execution, without undue communication overhead. If the accelerators are far away, for example across PCIe, the operation would incur substantial offload overhead multiple times, making the use of each accelerator less attractive. In the long run, the potential performance gains may justify additional software engineering effort in file formats to enable the exploitation of sequences of hardware accelerators; this is left to future work.

3.6 Window Size Requirements

A compression algorithm's window size determines the amount of recent history the algorithm will keep when searching for matches during LZ77-style de-duplication. Correspondingly, during decompression, the window size represents the maximum offset into the recently produced output from which a copy command can read data.

Our first algorithm of interest, Snappy, has a *fixed* window size of 64 KiB for compression and decompression [5]. For ZStd compression, Figure 5a shows the per-call fleet-wide window-size distribution. We see that slightly over 50% of bytes compressed by ZStd use a window size of 32 KiB or less. However, the upper 50% of the distribution quickly grows, with a 75^{th} percentile between 512 KiB and 1 MiB and tails as high as 16 MiB. The distribution for ZStd decompression is shown in Figure 5b, with a median of 1 MiB.

This parameter can affect accelerator design and performance. For compression, the window is commonly kept in SRAM, registers, or even expensive CAM structures. For decompression, the window is commonly kept in SRAM. However, beyond for example 32 KiB, on-chip storage can become prohibitively expensive. Notably, the existing state-of-the-art compression accelerator for a heavyweight algorithm, IBM's z15 compression accelerator [18], offers a window size of 32 KiB, meaning it would not be able to handle 50% of these compression calls in Google's fleet.

This further argues for a near-core accelerator with access to the memory hierarchy, which would allow the accelerator to "fall back" to accessing the history from the L2 cache or main memory. This design space is further explored in Section 6.

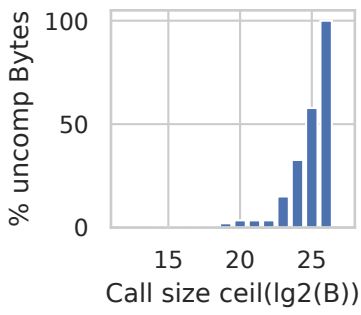


Figure 6: Call size distribution from four popular open-source compression benchmarks.

3.7 Do existing open-source compression benchmarks represent hyperscale requirements?

Several of our analyses thus far have motivated the need to perform a design-space exploration of (de)compression acceleration within the context of a complete system. However, performing such an exploration requires representative (de)compression benchmarks used as input to the accelerators.

Many benchmark suites exist that aim to provide a standard set of input files to evaluate compression algorithms. The most well-known of these is Silesia [33], which, for example, is used to provide the "default" results in the `README`s of both ZStd and `lzbench`, a common compression benchmarking tool. Other commonly used benchmarks (e.g., in [18]) include Canterbury [22], Calgary [27], and several benchmarks included with Snappy (we will refer to the collection as SnappyFiles) [6].

Unfortunately, we find that they are not representative of Google's fleet usage of (de)compression. As one dimension of comparison, we can bin these open benchmarks by call size as we did for fleet-wide compression calls in Figure 3. Figure 6 shows this call size distribution for open-source benchmarks, which we can see is vastly different from the fleet distribution. For example, the median call sizes of the distributions differ by an astounding 256×. More work is clearly needed to realistically evaluate compression in a hyperscale fleet. We will describe the construction of representative benchmarks in Section 4.

3.8 Key Cloud Provider Fleet Profiling Lessons for Hyperscale CDPUs

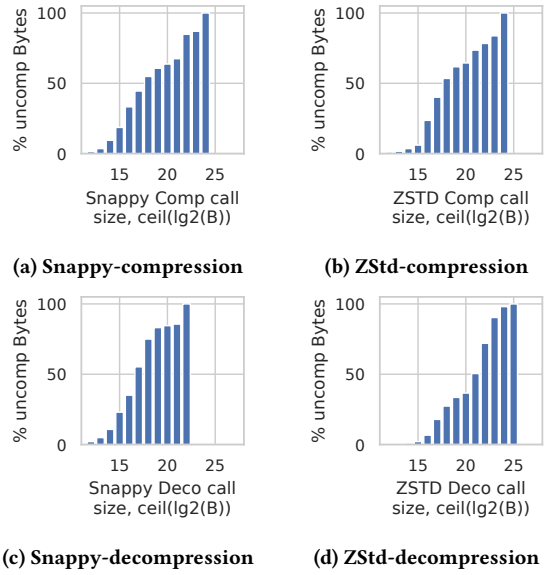
Before continuing, we summarize the key profiling insights gleaned thus far and highlight the important questions that remain:

(1) Significant headroom exists in fleet compression usage for accelerators that improve compression ratio vs. compute tradeoffs:
   (a) Lightweight algorithms dominate compression usage, handling 64% of compressed bytes.
   (b) Heavyweight algorithms are primarily used at lower compression levels: 88% of bytes compressed with ZStd are handled at level 3 (the default) or lower.
   (c) Services using heavyweight algorithms at high levels achieve higher compression ratios (1.35-1.97×), but at a significantly higher cost-per-byte (1.55-3.70×).
   (d) For many services, this increased CPU cost is untenable, presenting an opportunity for accelerators that achieve higher compression ratios within service performance bounds.
(2) Change in (de)compression algorithm usage in Google's fleet over time (e.g., ZStd's 0% → 10% of fleet (de)compression cycles in 1 year) aligns with agile hardware design cycles and motivates a re-usable CDPU *generator* over point designs.
(3) Fleet call sizes are not sufficiently biased towards small/large calls to immediately determine accelerator placement.
   (a) Instead, understanding placement requires design-space exploration of an implementation running representative benchmarks.
(4) Accelerator chaining between serialization and compression, which could ease placement requirements, is non-trivial.
   (a) At a minimum, chaining will require re-architecting file format libraries, which are responsible for invoking 49.2% of fleet (de)compression cycles, and the ability to maintain multiple contexts in the accelerator.
   (b) On the other hand, both of these concerns can be avoided while maintaining most chaining benefits if the accelerator is placed close to the CPU, with direct access to caches or main memory.
(5) History window sizes in the fleet are also insufficiently biased to make a clear recommendation for on-accelerator history window sizing.
   (a) Like accelerator placement, this will require design-space exploration of an accelerator implementation.
(6) Existing open-source (de)compression benchmarks used by prior work do not represent hyperscale compression usage.
   (a) For example, call size distributions differ greatly between open-source benchmarks and Google's fleet, even at a high-level; the median call size in popular open-source benchmarks is 256× the fleet's median call size.

While hyperscale fleet profiling has provided several insights about CDPU design requirements, a few critical questions remain that are difficult to explore without a concrete implementation evaluated in the context of a complete system. In the rest of this paper, we will build a parameterized CDPU generator and a hyperscale-representative (de)compression benchmark suite, then answer the open CDPU design questions by performing an extensive design-space exploration.

4 BUILDING OPEN-SOURCE HYPERSCALE-REPRESENTATIVE (DE)COMPRESSION BENCHMARKS

To produce (de)compression benchmarks representative of Google's fleet requirements, while preserving privacy in Google's datasets, we build an open-source (de)compression benchmark generator that produces representative benchmarks from summary statistics about a private data set. By supplying this generator with profiles of Google's fleet, we produce the open-source *HyperCompressBench*, a benchmark suite representative of (de)compression requirements in Google's fleet.



(a) Snappy-compression **(b) ZStd-compression**

(c) Snappy-decompression **(d) ZStd-decompression**

Figure 7: Call-size distributions for HyperCompressBench.

The generator starts by breaking all files from the Silesia, Canterbury, Calgary, and SnappyFiles benchmarks into fixed-size chunks. Each chunk is individually run through all combinations of supported algorithms and parameters (window size, compression level) to obtain a compression ratio for that chunk for each algorithm/parameters pair. This data is stored in lookup tables indexed by the compression ratio.

The generator then ingests metrics such as call size, compression ratio, window size, and compression level from the aforementioned fleet profiling data, constructs distributions from these metrics, and samples from the distributions to produce a set of target parameters for a single benchmark file.

For each such set of target parameters, the generator walks through the lookup table, greedily selecting chunks with the closest compression ratio and adding them to the output file until the target call size is reached. At various points during this process, the generator evaluates the file assembled so far and adjusts the target ratio accordingly. To avoid pathological sequences, random shuffles are introduced both within the lookup table and the output. The completed file is saved along with the parameters (level and window size) that should be applied when it is used. This process is repeated until we have a sufficient number of benchmark files to represent the overall distribution of calls across various dimensions. We find around 8,000 to 10,000 files to be a suitable number for this work.

The entire process is repeated for each algorithm/operation pair of interest, in our case, (Snappy and ZStd) × (Compress and Decompress). In the rest of the paper, we refer to this suite of around 35,000 generated files as *HyperCompressBench*.

4.1 HyperCompressBench validation

We validate the suite across the swath of previously discussed metrics. For example, consider the distributions for call size, shown in Figure 7. We can see that these line-up very well with the fleet distributions from Figure 3 and preserve the shape of each algorithm/operation pair's unique distribution, in stark contrast to the existing open-source benchmark suite call-size distributions. Between each pair of distributions, the only significant difference is in the largest size bins—this is because these call sizes represent an extremely small proportion of uncompressed fleet bytes and thus are unlikely to be included in an 8,000 to 10,000 benchmark sample. Comparing compression ratios, we find that on average for each suite, achieved compression ratios are within 5%-10% of fleet compression ratios. While elided due to space constraints, the distributions for compression level and window sizes are also extremely similar to the fleet distributions shown in Figures 2b and 5 respectively.

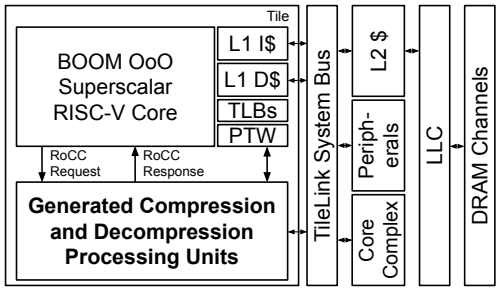


Figure 8: Top-level RISC-V SoC block diagram with CDPUs.

5 A PARAMETERIZED GENERATOR FOR COMPRESSION AND DECOMPRESSION PROCESSING UNITS (CDPUS)

Our open-source CDPU generator is implemented in Chisel RTL [25] and incorporated into the Chipyard RISC-V SoC generator ecosystem [21]. Figure 8 shows the overall architecture of the accelerated SoC, which is configured to use BOOM, an OoO superscalar RISC-V core with performance comparable to ARM A72-like cores [64].

The generated accelerators receive commands directly from the BOOM application core in the SoC via the RoCC interface [23], which allows the CPU to directly dispatch custom RISC-V instructions in its instruction stream to the accelerator within a few cycles. These *RoCC instructions* [23] can supply two 64-bit register values from the core to the accelerator. The accelerator accesses the same unified main memory space as the CPU through the 256 bit-wide TileLink-based NoC [38] and can issue memory requests with virtual addressing. As shown in Figure 8, all memory accesses made by the accelerator go through the L2 and LLC, which are shared with the application cores in the system.

Figures 9 and 10 show the block diagrams of complete decompressors and compressors respectively. Both handle the Snappy and ZStd algorithms. In these diagrams, components used by both algorithms are shown with a solid outline, components used only by Snappy with a dotted outline, and components used only by ZStd with a dashed outline. In the following subsections, we will outline the generator's library of reusable components used to build the aforementioned compressors and decompressors and give an overview of high-level parameters that can be modified.

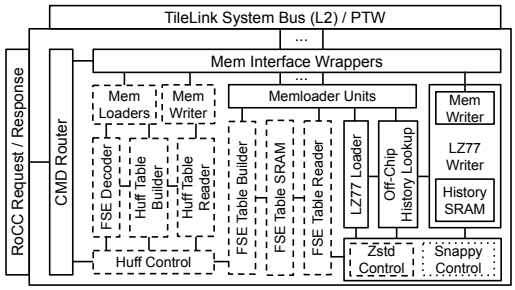


Figure 9: Block diagram for CDPU decompressor with support for Snappy and ZStd.

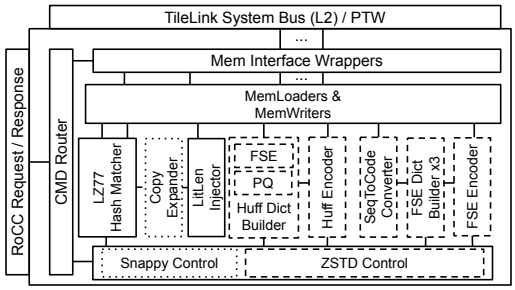


Figure 10: Block diagram for CDPU compressor with support for Snappy and ZStd.

5.1 System Interface Blocks

Our generator uses three types of blocks to interface accelerators to the rest of the system. *Memloaders* support streaming from the L2 cache, *Memwriters* support streaming to the L2 cache, and *CommandRouters* dispatch incoming commands to the appropriate subblocks. These are visible in both Figures 9 and 10.

5.2 LZ77 Decoder

The LZ77 Loader, Off-Chip History Lookup, and LZ77 Writer in Figure 9 comprise the LZ77 Decoder. This unit consumes sequences of `(offset, length, literal)` from a compressed input and produces the final stream of decompressed output data. The block primarily consists of a history window SRAM used to lookup matches based on offset and length, with the ability to fall back to making memory requests for matches that are further away than the configured size of the history SRAM.

5.3 Huffman Expander

The Huff Table Builder, Reader, and Control in Figure 9 comprise the Huffman Expander. Decoding Huffman-encoded streams is inherently serial because the starting position of a code cannot be known before decoding the previous code. The Huffman expander performs speculative decoding by issuing decode-table look-ups for a configurable number of starting bit positions, similar to the IBM z15 decompressor [18].

5.4 Finite-State Entropy (FSE) Expander

The FSE Expander (consisting of FSE Table Builder, SRAM, and Reader in Figure 9) first builds a decode table based on the normalized count statistics of each symbol by reading the input file stream. Then, the FSE expander reads the table to produce the decoded symbol, which is the sum of bits from the input file stream and the base value. The base value, number of bits to read from the input, and the next table entry to read are indicated in the table entry.

5.5 LZ77 Encoder

The LZ77 encoder (consisting of the LZ77 Hash Matcher and LitLen Injector blocks in Figure 10) performs streaming dictionary encoding of raw input data and produces output in the common `(offset, length, literal)` format. It primarily consists of a configurable hash table SRAM and a history window SRAM. This unit iterates over the data, checking the hash table for matches in the history and then checking the history buffer to find the extent of the match. If no match is available, the data is emitted as a literal.

5.6 Huffman Compressor

The Huffman compressor consists of two main modules, the Huffman dictionary builder and the Huffman encoder (Figure 10). The dictionary builder collects symbol statistics and writes the dictionary into memory. The encoder performs compression by performing look-ups into the dictionary builder.

5.7 Finite-State Entropy (FSE) Compressor

The FSE compressor is shown as part of Figure 10 and consists of three separate dictionary builders for each of literal length, match length, and offset and an FSE encoder that performs dictionary lookups to perform compression. The input stream is passed to the combinational SeqToCodeConverter which feeds the dictionary builders with the correct inputs while the encoder consumes the raw input stream.

5.8 Parameterization

Our framework supports two parameterization methods:

(1) Runtime configurable (`RunT`): These are parameters that can be changed after hardware is built, either for programmability or for rapid design space exploration.
(2) Compile-time configurable (`CompileT`): These are traditional hardware parameters that are fixed when the design is compiled.

The parameters available in our framework include:

5.8.1 CDPU-wide parameters:

(1) Accelerator placement (`CompileT`), including:
  (a) Near-core RoCC/on-NoC; no latency injection
  (b) Chiplet; 25ns latency injection
  (c) PCIeLocalCache: PCIe+DDIO, assuming PCIe card has large SRAM cache and on-board DRAM; 200ns latency injection (measurements from [48]) for raw input + final output, no latency injection for intermediate reads/writes
  (d) PCIeNoCache: PCIe+DDIO, assuming PCIe card does not have on-board cache/DRAM; 200ns latency injection for all requests
(2) Algorithm support (`RunT` & `CompileT`)

5.8.2 LZ77 decoder parameters:

(3) History Window Size (RunT & CompileT)

5.8.3 LZ77 encoder parameters:

(4) History Window Size (RunT & CompileT)
(5) Hash-table number of entries (RunT & CompileT)
(6) Hash-table associativity (RunT & CompileT)
(7) Hash-table contents (CompileT)
(8) Hash Function (CompileT)

5.8.4 Huffman expander parameters:

(9) Number of speculations (CompileT)

5.8.5 Huffman compressor parameters:

(10) Number of Bytes per cycle to collect symbol stats (CompileT)

5.8.6 FSE compressor parameters:

(11) Number of Bytes per cycle to collect symbol stats (CompileT)
(12) Max accuracy of FSE compression tables (CompileT)

6 CDPU DESIGN SPACE EXPLORATION

6.1 Evaluation Methodology

We perform design-space exploration (DSE) of our accelerated systems implemented in RTL running HyperCompressBench using FireSim [44], which provides high-performance, deterministic, and cycle-exact[4] modeling of designs, while cycle-accurately modeling I/O, including DRAM [28].

Each benchmark is run on two systems: a single-core RISC-V system with CDPUs attached, modeled at 2 GHz core/CDPU frequency, and one core (2 HT) of a Xeon E5-2686 v4-based server, running at 2.3 GHz base/2.7 GHz turbo.

Performance results for our accelerated systems are reported by measuring end-to-end operation time from the perspective of software (i.e. the time taken by an entire compression or decompression call, without overlapping requests). Performance results for the Xeon are collected using `lzbench` [55], a standard tool for in-memory (de)compression algorithm benchmarking. In HyperCompressBench, a suite's aggregate performance metric is the total amount of time required to (de)compress each benchmark file in the suite. Lastly, we report ASIC area estimates by pushing designs through synthesis [60] for a commercial 16nm-class process.

6.2 Snappy Decompressor

Figure 11 shows speedup and area results from a CDPU generated for Snappy Decompression, configured with a range of on-accelerator history windows (given on the x-axis) and in a variety of placements in the system. In this design, offsets beyond the on-accelerator SRAM fall back to the L2 cache. We see that the CDPU placed near-core (RoCC) with the largest on-accelerator window size (equal to Snappy's SW maximum of 64 KB), achieves the highest speedup; it is over 10× faster than the Xeon (11.4 GB/s accelerated vs. 1.1 GB/s Xeon), while consuming 0.431mm^2 of silicon area in 16nm. As a comparison, this is less than 2.4% of the area of a single modern Xeon Core Tile (17.98mm^2 in 14nm, reported in [63]). If we

[4]All components of the RISC-V SoC written in RTL, including our accel. design, are modeled bit-by-bit and cycle-by-cycle exactly as they would perform in silicon taped-out using the same RTL.

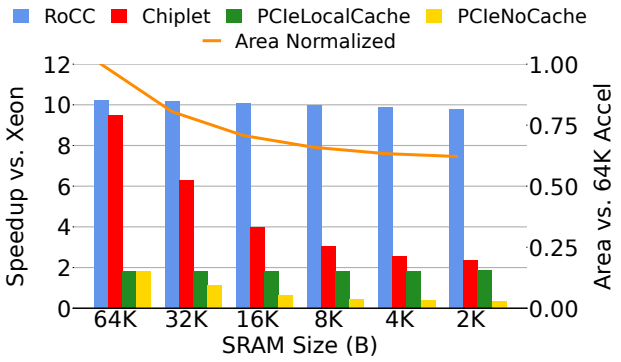


Figure 11: CDPU speedup running Snappy Decompression on HyperCompressBench across accelerator placements and History SRAM Sizes. Area is normalized vs. the 64KB history SRAM accelerator.

instead shrink the on-CDPU history to 2 KB, we find a potentially more fruitful design point: we can achieve a 38% reduction in area for only a 4.3% reduction in speedup (i.e., 9.8× speedup vs. Xeon while consuming 1.5% of the area).

As discussed in Section 5.8, we also model integrating the CDPU over PCIe+DDIO and re-run the sweep of on-accelerator SRAM size, which is shown by the "PCIeNoCache" series in Figure 11. Even with a 64K SRAM (no off-accelerator history lookups), we see that even the cost of loading/writing input/output data once over PCIe results in a significant (5.6×) slowdown vs. the near-core CDPU, due to the large number of small decompressions in the fleet (Fig. 3c).

The increased latency of PCIe also means that the accelerator cannot take advantage of the same performance vs. history SRAM-size tradeoff as the near-core accelerator: the PCIe-attached 32K SRAM design loses most of the performance advantage of the already degraded PCIe 64K design, and performance only degrades further from there. The "PCIeLocalCache" series in Figure 11 somewhat mitigates this by modeling a shared on-die SRAM cache and local DRAM attached to the PCIe card. In this situation, we can see that the SRAM optimization continues to work, albeit with an identical starting speedup (at the 64K size) as "PCIeNoCache".

Chiplet integration techniques and new protocols like CXL, UCIe, CCIX, and CAPI offer a new "intermediate" placement option for accelerators; accelerators can be manufactured in a separate die, reducing integration cost, while still remaining on the same package as the core. As discussed in Section 5.8, we can model this integration technique in our framework. The results of running the Snappy decompressor in this placement are shown in the "Chiplet" series in Figure 11. Considering the configuration with 64K history size, we can see that Chiplet integration is an attractive solution for a Snappy accelerator; it still achieves a 9.5× speedup vs. the Xeon, despite the added latency. However, we can see that performance suffers as more requests are forced to cross the Chiplet interconnect; at the smallest history window sizes, speedups drop such that they are on par with PCIe-based integration.

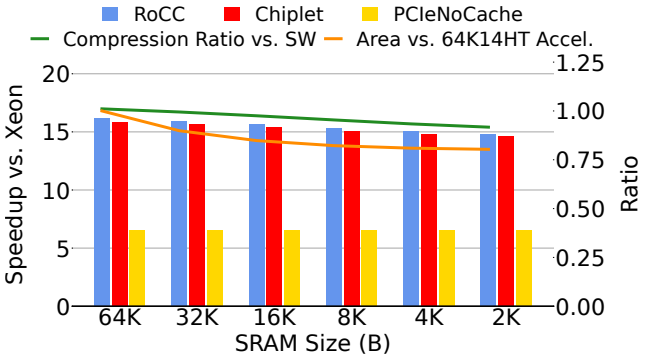


Figure 12: CDPU speedup/area running Snappy Compression on HyperCompressBench across CDPU placements and History SRAM Sizes. Area is norm-ed vs. the 64K history SRAM and 2^{14} hash table entry Snappy CDPU.

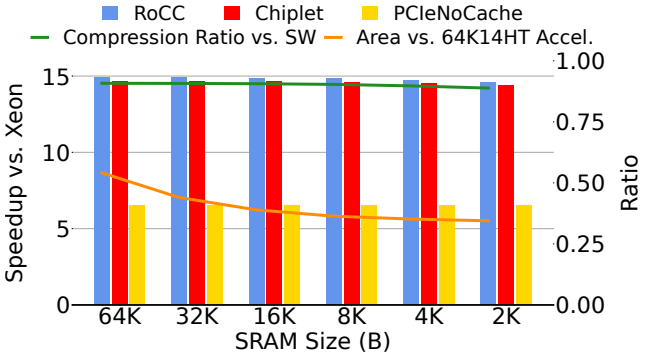


Figure 13: CDPU speedup/area running Snappy Compression on HyperCompressBench across CDPU placements and History SRAM Sizes, with only 2^9 Hash Table Entries. Area is norm-ed vs. the 64K history SRAM and 2^{14} hash table entry Snappy CDPU.

6.3 Snappy Compressor

Figure 12 shows speedup, compression ratio, and area results for the Snappy Compress accelerator, covering a range of on-accelerator history windows (on the x-axis). Area results are normalized to the largest version of the accelerator, which has a 64K History SRAM and 2^{14} hash table entires ("64K14HT" on plots). This design consumes 0.851 mm^2 in a 16nm process or about 4.7% the area of a Xeon Core [63]. Reducing the history SRAM size restricts the maximum matching offset that can be identified, and large offset matching does not fall back to the L2 cache since history checking is necessarily serial in compression. Interestingly, the 64 KB SRAM design achieves a 1.1% *higher* compression ratio than Snappy SW. This is because the software implements a skipping mechanism that avoids hash-table lookups when data appears incompressible to save cycles. In a hardware implementation, this optimization is not useful. Therefore, the accelerator has more "chances" to find a match than SW. As the SRAM size is reduced, we do see a drop-off in the achieved compression ratio as compared to software, ranging from an 8% loss at 2 KB (with 20% area savings) to a 0.5% loss at 32 KB (with 10% area savings).

We also see that across the swath of history window sizes, the accelerator achieves significant speedup compared to the Xeon. For example, the 64 KB configuration achieves over 16× speedup compared to the Xeon (5.84 GB/s accel. vs. 0.36 GB/s Xeon). The various smaller configurations achieve between 14.8× and 15.5× speedup, losing performance only because of the increased amount of data they must write due to the lower achieved compression ratio.

Figure 12 also shows various compression accelerator placements. We see again that a Chiplet-integrated design performs very well, achieving less than 1.7% loss of speedup vs. the near core design across the swath of SRAM sizes. PCIe again struggles, but fares much better than in the decompression case, with speedups shrinking to around 6.6×. Note that PCIeNoCache and PCIeLocalCache are identical for compression, given that there are no intermediate data accesses.

Given that Snappy is a lightweight algorithm, we can ask an interesting question: how small of a Snappy accelerator can we build while still achieving meaningful compression and high performance? In Figure 12 we can see that reducing the history window size to 2K for compression can result in negligible loss of speedup and a small, but potentially tolerable 8% loss in compression ratio, while reducing accelerator area by 20%. Figure 13 shows the results of tuning another design knob: the number of hash table entries. Reducing the number of entries increases the likelihood of collisions and reduces the chance of finding optimal matches in the history window. However, we can see that reducing the number of hash table entries can provide drastic area wins: a snappy compression accelerator with 2^9 hash table entires and a 2K history SRAM consumes only 34% of the area of the full-size design (and only 1.6% of the area of a Xeon Core), with a negligible loss of speedup and while only increasing compression ratio loss by 3% compared to the 2K history, 2^{14} hash table entry design.

6.4 ZStd Decompressor

Figure 14 shows speedup and area results from a CDPU generated for ZStd Decompression, configured with a range of on-accelerator history windows (given on the x-axis) and in a variety of placements in the system. The largest design in this plot (64K SRAM) achieves 4.2× speedup vs. the Xeon (3.95 GB/s accelerated vs. 0.94 GB/s Xeon).

We can see that overall performance is reduced compared to the Snappy accelerator. While this is not directly comparable since the Snappy/ZStd suites in HyperCompressBench are different, we can broadly see the cost of the additional entropy decoding steps on the accelerator's performance, especially since the LZ77 decoding block is re-used between Snappy and ZStd accelerators. This added cost attenuates both the area savings and performance impact of reducing history SRAM compared to the Snappy decompressor; the overall savings moving from the 64K SRAM design (1.9 mm^2 in 16nm) to the 2K SRAM design of the ZStd compressor is only 8.6%.

An additional parameter that can be swept in the ZStd decompressor as compared to Snappy is the amount of speculation allowed

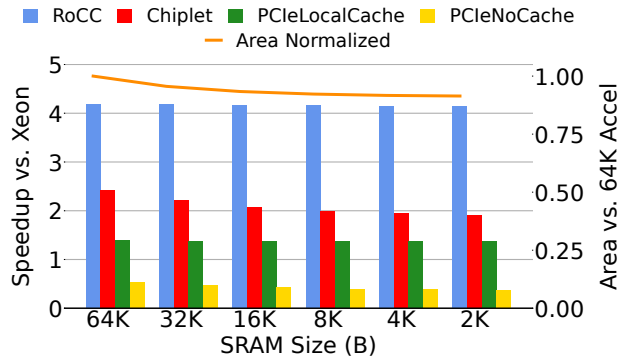


Figure 14: CDPU speedup running ZStd Decompression on HyperCompressBench across accelerator placements and History SRAM Sizes. Area is normalized vs. the 64KB history SRAM accelerator.

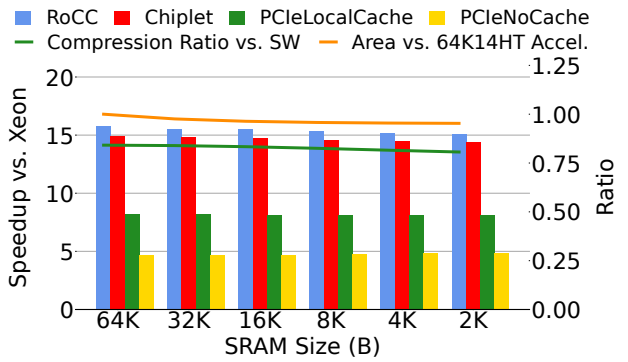


Figure 15: CDPU speedup/area running ZStd Compression on HyperCompressBench across CDPU placements and History SRAM Sizes. Area is norm-ed vs. the 64K hist. and 2^{14} hash table entry ZStd CDPU.

in the Huffman Decoder. All results in Figure 14 used a speculation of 16. To better understand the design space, we explored two additional speculation design points: 32 (similar to IBM z15) and 4 (as a minimum reasonable design point), while keeping history SRAM size fixed at 64K. The 32 speculation design increases speedup over Xeon to 5.64×, while requiring an additional 18% area as compared to the 16 speculation design. The 4 speculation design reduces speedup over Xeon to 2.11×, while requiring 10% less area as compared to the 16 speculation design. As we can see, for the ZStd decompressor, tuning the speculation amount produces a much larger swing in design quality-of-result than history SRAM size.

6.5 ZStd Compressor

Figure 15 shows speedup, compression ratio, and area results for the ZStd Compress accelerator, covering a range of on-accelerator history windows (on the x-axis). Area results are normalized to the largest version of the accelerator, which has a 64K History SRAM and 2^{14} hash table entries ("64K14HT" on plots). This design consumes 3.48mm^2 in a 16nm process. As this accelerator re-uses the LZ77 encoder block from the Snappy accelerator, restricting history SRAM size similarly restricts the maximum matching offset that can be identified. Looking first at compression ratio, we see that the accelerator achieves only 84% of the compression ratio of software, likely primarily due to the fact that we are re-using the LZ77 encoder block as configured for Snappy. We leave exploring more complicated LZ77 encoding techniques to future work. With the caveat that compression ratio is reduced, the largest configuration of accelerator achieves a 15.8x speedup compared to the Xeon (3.5 GB/s accelerated vs. 0.22 GB/s Xeon).

6.6 Key Implementation-Based Design-Space Exploration Lessons for Hyperscale CDPUs

Our design space exploration shows the importance of focusing not only on the microarchitectural design of CDPUs, but also their high-level parameters. By tuning these high-level parameters in the previous section, we observed for example, 46× differences in speedups and 66% savings in silicon area. Here, we summarize our key findings:

(1) Decompression accelerator feasibility is very heavily affected by accelerator placement. Given data sizes observed in Google's fleet, near-core accelerators (10× speedup for Snappy, 4× speedup for ZStd) perform over 3 to 5.6 *times* better than PCIe attached accelerators (1.8× speedup for Snappy, 1.4× speedup for ZStd). Chiplets offer a reasonable middle ground for Snappy, with our chiplet-integrated accelerator (9.5× speedup) performing only 1.1× worse than the near-core accelerator.
(2) In contrast, compression is less sensitive to accelerator placement; we observe over 6.6× speedup (Snappy) or 8.2× speedup (ZStd) in the PCIe attached cases. However, the biggest performance gains are still seen for near-core and chiplet-integrated designs (around 15 to 16× speedup for both Snappy and ZStd).
(3) Snappy decompression accelerator area is dominated by history size, which also affects speedup (but not compression ratio). Given data characteristics in Google's fleet, a 38% silicon area savings can be achieved by slightly sacrificing speedup (9.8× vs. 10× speedup).
(4) ZStd decompression accelerator area is dominated by varying the amount of speculation in the Huffman stage. Given data characteristics in Google's fleet, there is a 31% silicon area cost increase between speculation amounts of 4 and 32, but this comes with a significant corresponding improvement in speedup (2.1× vs. 5.6× speedup).
(5) Snappy compression accelerator area is dominated by history buffer size and hash table size. When both are reduced, a negligible sacrifice in speedup and a 12% sacrifice in compression ratio can result in reducing accelerator silicon area by 66%.

7 RELATED WORK

A few prior studies have presented (de)compression metrics as part of broader hyperscaler fleet characterizations [40, 41, 56]. Our study is the first to take a fleet-wide, multi-year deep-dive into (de)compression usage at a major cloud provider by profiling Google's fleet and derives several novel insights for CDPU design.

Furthermore, we use the insights gained to build a parameterized generator for CDPUs that supports hyperscale use-cases, and translate our profiling data into open-source, hyperscaler-representative (de)compression benchmarks that can be used by the community.

Many prior studies have explored implementing hardware accelerators for lossless block-level (de)compression, both in academia [29, 50], industrial research [36, 53], and commercial products [10–14, 18, 58]. However, all of these studies only explore a single point in the design space; they focus on a single algorithm (usually Flate or ZStd), in a single placement (PCIe, NoC-attached, on-chipset, etc.), and sometimes only a single direction (decompress or compress). Furthermore, these studies usually run existing open-source benchmarks, which, as shown in Section 4, are not representative of hyperscale workloads. To our knowledge, we are the first study to build a highly-parameterized CDPU *generator* that supports multiple algorithms using a common set of high-performance, re-usable primitives. Our generator integrates into a RISC-V SoC framework that allows for rapid evaluation of CDPUs across system placements and configuration parameters. Furthermore, we evaluate our generated designs with HyperCompressBench, a (de)compression benchmark that is representative of hyperscale workloads.

However, for our design space evaluations to produce realistic results, it is important to contextualize and validate our observed results with those published in prior studies. To that end, we compare against the current state of the art, the NXU accelerator for the IBM POWER9 and z15 [18]. While the NXU study does not provide a directly comparable performance result using open-source benchmarks, we can extrapolate from its performance vs. data size plots and the size distribution of input files in HyperCompressBench. This calculation projects performance of the NXU on HyperCompressBench of 5.6 to 7.1 GB/s for compression and 6.7 to 7.7 GB/s for decompression. Our results for compression (5.8 GB/s Snappy, 3.5 GB/s ZStd) and decompression (11.4 GB/s Snappy, 5.3 GB/s ZStd) are comparable, given our RISC-V SoC's weaker memory system and algorithmic differences. In area terms, our academic prototype is similar, but could benefit from greater tuning/engineering effort, with our design consuming around 1.3 mm^2 (Snappy) or 5.7 mm^2 (ZStd) in a 16nm process, while the IBM NXU consumes around 3.5 mm^2 in the GF14 process (extrapolated from [18, 62]).

To our knowledge, the state-of-the-art *open-source* compression and decompression implementation is Project Zipline [13, 30, 32, 58] from Microsoft, which has also been fabricated in the Corsica ASIC [30, 59]. The ASIC version of this design is limited to 25 Gbps for single requests (3.125 GB/s) [30]. Also in contrast, our design is heavily parameterized, supporting several compile-time and run-time configurable parameters, and is integrated into a complete system for evaluation.

FPGAs have also been proposed as a host platform for compression and decompression accelerators constructed using handwritten RTL or high-level synthesis tools [29, 36, 46, 50]. Unfortunately, FPGAs as a basis technology are insufficiently performant to support (de)compression as compared to ASIC designs—our generated accelerators are significantly faster than the state-of-the-art handwritten [29] and HLS-generated [46] FPGA-hosted implementations. In Section 3.4, we also demonstrated that the flexibility of FPGAs is unnecessary for the pace of (de)compression algorithm evolution in WSCs. Lastly, the Corsica ASIC's compression engine has also been shown to achieve improved performance over FPGA-hosted solutions [30].

Several interesting industrial products are also on the horizon, including NVIDIA's DPU [17], Intel's IPU [15], and Intel Sapphire Rapids/QAT [16]. At time of writing, little commercial benchmarking data is available publicly for these systems.

8 CONCLUSION

In this work, we presented a detailed fleet-wide characterization of (de)compression usage at a major cloud provider by profiling Google's datacenter fleet. We showed that (de)compression consumes significant fleet CPU cycles, even though services underutilize the most aggressive forms of compression, presenting an opportunity for hardware acceleration to save resources beyond merely CPU cycles.

We then presented the first end-to-end design/evaluation framework for CDPUs, including: 1. An open-source RTL-based CDPU generator that supports many run-time and compile-time parameters. 2. Integration into an open-source RISC-V SoC for rapid performance and silicon area evaluation with varying CDPU placements and configurations. 3. An open-source (de)compression benchmark, HyperCompressBench, that represents (de)compression usage in Google's fleet.

While a large body of prior work has improved the microarchitectural state-of-the-art for CDPUs supporting various algorithms in fixed contexts [10–18, 29, 30, 32, 36, 46, 50, 53, 58, 59], we found that higher-level design parameters like accelerator placement, hash table sizing, history window sizes, and more are as critical when considering the feasibility of CDPU integration, but were previously not well-studied in the literature.

Using our CDPU design framework, we performed an extensive design space exploration running HyperCompressBench. Our design-space exploration spanned a 46× range in accelerator speedup, 3× range in silicon area (for a single pipeline), and explored a variety of accelerator integration techniques to better understand optimal CDPU designs for hyperscale contexts. Our final hyperscale-optimized accelerator instances are up to 10× to 16× faster than a single Xeon core, while consuming a small fraction (as little as 2.4% to 4.7%) of the area.

ACKNOWLEDGMENTS

This work builds on profiling infrastructure work done by several engineering teams at Google (e.g., GWP) and we would like to thank our colleagues in those teams, including Todd Jackson, Garrett Wang, and Alexey Alexandrov. We would also like to thank Daniel Berlin, Jichuan Chang, Chris Kennelly, and the anonymous reviewers and artifact evaluators for their valuable feedback. The information, data, or work presented herein was funded in part by SLICE Lab industrial sponsors and affiliates and by NSF CCRI ENS Chipyard Award #2016662 and by NSF Award CCF-1955450. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact appendix describes how to reproduce the CDPU Design Space Exploration results from Section 6 of this paper. As in Section 6, we will use FireSim FPGA-accelerated simulations to cycle-exactly simulate the entire RISC-V SoC containing the RTL implementations of CDPUs (compression and decompression accelerators). We will run HyperCompressBench, the benchmark suite we created from fleet-wide profiling at Google, on both a Xeon system (for the baseline) and our RISC-V SoC augmented with CDPUs. We will sweep the design parameters explored in Section 6 to collect accelerator performance metrics and reproduce the accelerator trade-offs and insights discussed in the paper.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** AWS FPGA Developer AMI 1.12.1.
- **Hardware:** AWS EC2 instances: 1× `c5.9xlarge`, 16× `f1.2xlarge`, 1× `m4.large`.
- **Metrics:** Compression and decompression throughput (GB/s), compression ratio.
- **Output:** Compression and decompression performance and compression ratio plots. HyperCompressBench call size distribution plots. Regeneration of paper text that contains data.
- **Experiments:** FireSim simulations of compression and decompression accelerators incorporated into a RISC-V SoC, running HyperCompressBench.
- **How much disk space is required?:** 2000GB (on EC2 instance).
- **How much time is needed to prepare workflow?:** 1 hour (scripted installation).
- **How much time is needed to complete experiments?:** 6 hours for Snappy, 110 hours for ZStd (both fully-automated).
- **Publicly available:** Yes.
- **Code licenses:** Several, see download.
- **Archived:** https://doi.org/10.5281/zenodo.7812634, https://doi.org/10.5281/zenodo.7812577, https://doi.org/10.5281/zenodo.7812573, https://doi.org/10.5281/zenodo.7812563.

A.3 Description

A.3.1 How to access. The artifact consists of four git repositories preserved on Zenodo.

(1) **`chipyard-compress-acc-ae`**: Chipyard RISC-V SoC generation environment, customized for CDPU evaluation. Zenodo: https://doi.org/10.5281/zenodo.7812634
(2) **`firesim-compress-acc-ae`**: FireSim simulation environment, customized for CDPU evaluation. Zenodo: https://doi.org/10.5281/zenodo.7812577
(3) **`compress-acc-ae`**: Compression and decompression accelerator implementation (RTL), software, and scripts. Zenodo: https://doi.org/10.5281/zenodo.7812573
(4) **`HyperCompressBench`**: Compression and decompression benchmarks representative of (de)compression usage in Google's datacenter fleet, created and open-sourced for this paper. Zenodo: https://doi.org/10.5281/zenodo.7812563

Users need not download the latter three repositories manually—they will be obtained automatically from Zenodo when the first repository is set up in the next section.

A.3.2 Hardware dependencies. One AWS EC2 `c5.9xlarge` instance (also referred to as the "manager" instance), sixteen `f1.2xlarge` instances, and one `m4.large` instance are required. The latter two instance types will be launched automatically by FireSim's manager.

To optionally run FPGA builds (see Appendix A.7.2), seven additional `z1d.6xlarges` are required, however we provide pre-built FPGA images to avoid the long latency (≈ 18 hours) of this process.

A.3.3 Software dependencies. Installing `mosh` (https://mosh.org/) on your local machine is highly recommended for reliable access to EC2 instances. All other requirements are automatically installed by scripts in the following sections.

A.4 Installation

First, follow the instructions on the FireSim website[5] to create a manager instance on EC2. You must complete up to and including "Section 1.3.1.2: Key Setup, Part 2", with the following changes in "Section 1.3.1":

(1) When instructed to launch a `c5.4xlarge` instance, choose a `c5.9xlarge` instead.
(2) When entering the root EBS volume size, use 2000GB rather than 300GB.

Once you have completed up to and including "Section 1.3.1.2" in the FireSim docs, you should have a manager instance set up, with an IP address and key. Use either `ssh` or `mosh` to login to the instance.

```
# Option 1: USE SSH
$ ssh -i KEY.pem centos@IP_ADDR
# Option 2: USE MOSH
$ mosh --ssh="ssh -i KEY.pem" centos@IP_ADDR
```

From this point forward, all commands should be run on the manager instance.

If using `ssh`, be sure to start `screen` or `tmux` on the manager so that the artifact continues running even if your network connection is interrupted.

Begin by fetching the top-level repository from Zenodo, like so:

```
$ cd $HOME
# Enter as a single line:
$ curl -Ls -w %{url_effective} -o a https://doi.org/
       10.5281/zenodo.7812634 > DL_url
$ wget $(cat DL_url)/files/chipyard-compress-acc-ae.zip
$ unzip chipyard-compress-acc-ae.zip
```

Next, run the following, which will initialize all dependencies and run basic Chipyard and FireSim setup steps (RISC-V toolchain installation, host toolchain installation, etc.):

```
$ cd chipyard-compress-acc-ae
$ ./scripts/first-clone-setup-fast.sh
```

This step should take around 45 minutes. Upon successful completion, it will print:

```
first-clone-setup-fast.sh complete.
```

Once this is complete, run:

[5] https://docs.fires.im/en/1.15.2/Initial-Setup/index.html

```
$ source env.sh
$ cd sims/firesim
$ source sourceme-f1-manager.sh
```

Finally, run the following to finish setting up FireSim. You can enter your email address when prompted if you plan to run the optional FPGA builds in Appendix A.7.2, otherwise just hit Enter.

```
$ firesim managerinit
```

Now, your manager instance is fully set up to run CDPU sims.

A.5 Experiment workflow

Now that our manager is set up, we will run the full artifact evaluation script, which will automatically do the following:

(1) On the manager instance, extract HyperCompressBench and compile RISC-V/CDPU benchmarks for the ≈35,000 benchmark files we need.
(2) For isolated Xeon baseline runs, launch an `m4.large`, run HyperCompressBench on it using `lzbench`, collect results, and terminate the `m4.large`.
(3) On the manager instance, build FireSim host-side drivers required to drive each FPGA-accelerated simulation.
(4) Launch sixteen `f1.2xlarge` instances, which provide a total of sixteen FPGAs to run simulations on in parallel.
(5) Run FireSim simulations, repeating the following for the 16 workloads of interest:
    (a) Copy all simulation infrastructure to the F1 instances.
    (b) Run the set of benchmarks on 16 simulated systems in parallel (one `f1.2xlarge` has 1 FPGA).
    (c) Copy results back to the manager instance.
(6) Terminate the sixteen `f1.2xlarge` instances.
(7) On the manager, re-generate accelerator performance plots from this paper (and sections of the paper text that use this data), using data collected from your runs.

Note that this script will *not* rebuild FPGA images for the system by default, since each build takes around 18 hours. We instead provide pre-built images by default (see `$COMPRESSACC_FSIM/config_others/config_hwdb.yaml`). If you would like to build your own images, see Appendix A.7.2, then return here.

Now, run the aforementioned full artifact evaluation script:

```
$ cd $COMPRESSACC_FSIM
$ ./run-ae-full.sh
```

This takes around 6 hours for Snappy and 110 hours for ZStd. When complete, it will print:

```
run-ae-full.sh complete.
```

The FireSim manager will automatically terminate any instances it launched during this process.

A.6 Evaluation and expected results

Next, we will step through the plots generated from your run of `run-ae-full.sh` in the previous section. The following results generated from your run will be located in the `$HYPER_RESULTS` directory:

(1) Figures 7a, 7b, 7c, and 7d: `[Snappy,ZSTD]-[C,D]-callsizes.pdf`
(2) Figure 11: `snappy-decompression.pdf`
(3) Figure 12: `snappy-compression-ht14.pdf`
(4) Figure 13: `snappy-compression-ht9.pdf`
(5) Figure 14: `zstd-decompression.pdf`
(6) Figure 15: `zstd-compression-ht14.pdf`
(7) `FINAL_TEXT_SUMMARIES.txt` contains paper text re-generated with data obtained from these simulations. This excludes the single ZStd-Decomp-32spec data point, which requires ≈100 additional machine-days of software simulation.
(8) Raw results are located in the five `*.csv` files

A.7 Experiment customization

A.7.1 Customizing the design. Since the compression and decompression accelerators are written in Chisel RTL, incorporated into the Chipyard RISC-V SoC generator ecosystem, and modeled at high-performance using FireSim, they can be experimented with in a wide-variety of contexts, including in multi-core systems, attached to various kinds of processors, and with different memory hierarchy configurations, to name a few. These parameters are too numerous to list here; see the FireSim docs[6], Chipyard docs[7], and tutorial slides[8] for these configuration options.

The compression and decompression accelerator RTL is located in the `$COMPRESSACC_SRC` directory and can be customized as necessary, including using the runtime and compile-time configurable parameters outlined in Section 5.8, several of which we swept in this artifact evaluation.

A.7.2 Rebuilding FPGA images. We provide pre-built FPGA images for designs in this paper (generated from the included RTL), encoded in the configuration files in the artifact.

Rebuilding the supplied FPGA images can also be done by running `./buildafi.sh` in the `$COMPRESSACC_FSIM` directory. This will take around 18 hours, require seven `z1d.6xlarge` instances, generate seven new AGFIs (i.e., FPGA bitstreams on EC2 F1), and place their `config_hwdb.yaml` entries in `$BUILT_HWDB_ENTRIES/[config name]`. To use the new AGFIs, replace existing entries in the `$COMPRESSACC_FSIM/config_others/config_hwdb.yaml` file (or, for a new config, add it).

When an FPGA build completes, the FireSim manager will automatically terminate the instances it launched during the build process. More details about the FireSim FPGA build process can be found in the FireSim docs[9]. Note that many of the FireSim manager build configuration files are in a non-standard location to simplify scripting for artifact evaluation. Open `buildafi.sh` to see their locations.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

[6] https://docs.fires.im/en/1.15.2/
[7] https://chipyard.readthedocs.io/en/1.8.1/
[8] https://fires.im/asplos-2023-tutorial/
[9] https://docs.fires.im/en/1.15.2/Building-a-FireSim-AFI.html

REFERENCES

[1] [n. d.]. Brotli compression format. https://github.com/google/brotli.

[2] [n. d.]. FiniteStateEntropy: New Generation Entropy coders. https://github.com/Cyan4973/FiniteStateEntropy.

[3] [n. d.]. Gipfeli, a high-speed compression library. https://github.com/google/gipfeli.

[4] [n. d.]. LZO real-time data compression library. http://www.oberhumer.com/opensource/lzo/.

[5] [n. d.]. Snappy: A fast compressor/decompressor. https://github.com/google/snappy.

[6] [n. d.]. Snappy Testdata. https://github.com/google/snappy/tree/main/testdata.

[7] [n. d.]. zlib Home Site - A Massively Spiffy Yet Delicately Unobtrusive Compression Library. https://www.zlib.net/.

[8] [n. d.]. Zstandard - Real-time data compression algorithm. https://facebook.github.io/zstd/.

[9] 2011. Snappy compressed format description. https://github.com/google/snappy/blob/main/format_description.txt.

[10] 2013. Scaling Acceleration Capacity from 5 to 50 Gbps and Beyond with Intel QuickAssist Technology. https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/scaling-acceleration-capacity-brief.pdf.

[11] 2015. Intel QuickAssist Adapter 8950. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/quickassist-adapter-8950-brief.pdf.

[12] 2017. Product Brief: Intel Atom C3000 Processor. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/atom-c3000-family-brief.pdf.

[13] 2021. Project Zipline. https://github.com/opencomputeproject/Project-Zipline.

[14] 2022. AHA374 / AHA378 PCI Express Compression and Decompression Accelerator Card. http://www.aha.com/Uploads/aha374-378_brief_rev_c1.pdf.

[15] 2023. Intel Infrastructure Processing Unit (Intel IPU). https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html.

[16] 2023. Intel Launches 4th Gen Xeon Scalable "Sapphire Rapids". https://www.phoronix.com/image-viewer.php?id=intel-xeon-sapphire-rapids-max&image=intel_sapphirerapids_8_lrg.

[17] 2023. NVIDIA BlueField Data Processing Units. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[18] Bulent Abali, Bart Blaner, John Reilly, Matthias Klein, Ashutosh Mishra, Craig B. Agricola, Bedri Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J. Starke, Haren Myneni, and Charlie Wang. 2020. Data Compression Accelerator on IBM POWER9 and z15 Processors : Industrial Product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. https://doi.org/10.1109/ISCA45697.2020.00012

[19] Jyrki Alakuijala, Evgenii Kliuchnikov, Zoltan Szabadka, and Lode Vandevenne. 2015. Comparison of Brotli, Deflate, Zopfli, LZMA, LZHAM and Bzip2 Compression Algorithms. https://cran.r-project.org/web/packages/brotli/vignettes/brotli-2015-09-22.pdf.

[20] Jyrki Alakuijala and Zoltan Szabadka. 2016. Brotli Compressed Data Format. RFC 7932. https://doi.org/10.17487/RFC7932

[21] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (jul 2020), 10–21. https://doi.org/10.1109/MM.2020.2996616

[22] R. Arnold and T. Bell. 1997. A corpus for the evaluation of lossless compression algorithms. In *Proceedings DCC '97. Data Compression Conference*. 201–210. https://doi.org/10.1109/DCC.1997.582019

[23] Krste Asanović, Rimas Avižienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator*. Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley.

[24] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 462–473. https://doi.org/10.1145/3307650.3322234

[25] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*. 1212–1221. https://doi.org/10.1145/2228360.2228584

[26] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition. *Synthesis Lectures on Computer Architecture* 13, 3 (2018), i–189. https://doi.org/10.2200/S00874ED3V01Y201809CAC046

[27] Timothy C. Bell, John G. Cleary, and I. H. Witten. 1990. *Text compression*. Prentice Hall, Englewood Cliffs, N.J.

[28] David Biancolin, Sagar Karandikar, Donggyu Kim, Jack Koenig, Andrew Waterman, Jonathan Bachrach, and Krste Asanovic. 2019. FASED: FPGA-Accelerated Simulation and Evaluation of DRAM. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 330–339. https://doi.org/10.1145/3289602.3293894

[29] Jianyu Chen, Maurice Daverveldt, and Zaid Al-Ars. 2021. FPGA Acceleration of Zstd Compression Algorithm. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 188–191. https://doi.org/10.1109/IPDPSW52791.2021.00035

[30] Derek Chiou, Eric Chung, and Susan Carrie. 2019. HotChips31 Tutorial: (Cloud) Acceleration at Microsoft. https://old.hotchips.org/hc31/HC31_T2_Microsoft_CarrieChiouChung.pdf.

[31] Yann Collet and Murray Kucherawy. 2021. Zstandard Compression and the 'application/zstd' Media Type. RFC 8878. https://doi.org/10.17487/RFC8878

[32] Microsoft Corporation and Broadcom Corporation. 2019. Project Zipline Top Micro Architecture Specification. https://github.com/opencomputeproject/Project-Zipline/blob/master/specs/Project_Zipline_Top_Micro_Architecture_Specification.docx.

[33] Sebastian Deorowicz. [n. d.]. Silesia compression corpus. https://sun.aei.polsl.pl/~sdeor/index.php?page=silesia.

[34] L. Peter Deutsch. 1996. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951. https://doi.org/10.17487/RFC1951

[35] Jarek Duda, Khalid Tahboub, Neeraj J. Gadgil, and Edward J. Delp. 2015. The use of asymmetric numeral systems as an accurate replacement for Huffman coding. In *2015 Picture Coding Symposium (PCS)*. 65–69. https://doi.org/10.1109/PCS.2015.7170048

[36] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. 2015. A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 52–59. https://doi.org/10.1109/FCCM.2015.46

[37] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

[38] SiFive Inc. 2019. SiFive TileLink Specification. https://sifive.cdn.prismic.io/sifive%2Fcab05224-2df1-4af8-adee-8d9cba3378cd_tilelink-spec-1.8.0.pdf.

[39] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W. Lee. 2020. A Specialized Architecture for Object Serialization with Applications to Big Data Analytics. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 322–334. https://doi.org/10.1109/ISCA45697.2020.00036

[40] Geonhwa Jeong, Bikash Sharma, Nick Terrell, Abhishek Dhanotia, Zhiwei Zhao, Niket Agarwal, Arun Kejariwal, and Tushar Krishna. 2023. Characterization of Data Compression in Datacenters. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[41] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-Scale Computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 158–169. https://doi.org/10.1145/2749469.2750392

[42] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. 2017. Mallacc: Accelerating Memory Allocation. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 33–45. https://doi.org/10.1145/3037697.3037736

[43] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. 2021. A Hardware Accelerator for Protocol Buffers. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 462–478. https://doi.org/10.1145/3466752.3480051

[44] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. 2018. FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 29–42. https://doi.org/10.1109/ISCA.2018.00014

[45] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 317–330. https://doi.org/10.1145/3297858.3304053

[46] Morgan Ledwon, Bruce F. Cockburn, and Jie Han. 2020. High-Throughput FPGA-Based Hardware Accelerators for Deflate Compression and Decompression Using High-Level Synthesis. *IEEE Access* 8 (2020), 62207–62217. https://doi.org/10.1109/ACCESS.2020.2984191

[47] Rastislav Lenhardt and Jyrki Alakuijala. 2012. Gipfeli - High Speed Compression Algorithm. In *Proceedings of the 2012 Data Compression Conference (DCC '12)*. IEEE Computer Society, USA, 109–118. https://doi.org/10.1109/DCC.2012.19

[48] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 327–341. https://doi.org/10.1145/3230543.3230560

[49] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. 2020. Optimus Prime: Accelerating Data Transformation in Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1203–1216. https://doi.org/10.1145/3373376.3378501

[50] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. 2018. High-Throughput Lossless Compression on Tightly Coupled CPU-FPGA Platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 37–44. https://doi.org/10.1109/FCCM.2018.00015

[51] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, Anna Cheung, In Suk Chong, Niranjani Dasharathi, Jia Feng, Brian Fosco, Samuel Foss, Ben Gelb, Sara J. Gwin, Yoshiaki Hase, Da-ke He, C. Richard Ho, Roy W. Huffman Jr., Elisha Indupalli, Indira Jayaram, Poonacha Kongetira, Cho Mon Kyaw, Aaron Laursen, Yuan Li, Fong Lou, Kyle A. Lucke, JP Maaninen, Ramon Macias, Maire Mahony, David Alexander Munday, Srikanth Muroor, Narayana Penukonda, Eric Perkins-Argueta, Devin Persaud, Alex Ramirez, Ville-Mikko Rautio, Yolanda Ripley, Amir Salek, Sathish Sekar, Sergey N. Sokolov, Rob Springer, Don Stark, Mercedes Tan, Mark S. Wachsler, Andrew C. Walton, David A. Wickeraad, Alvin Wijaya, and Hon Kwan Wu. 2021. Warehouse-Scale Video Acceleration: Co-Design and Deployment in the Wild. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 600–615. https://doi.org/10.1145/3445814.3446723

[52] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro* (2010), 65–79. http://www.computer.org/portal/web/csdl/doi/10.1109/MM.2010.68

[53] Sudhir Satpathy, Vikram Suresh, Raghavan Kumar, Vinodh Gopal, James Guilford, Mark Anders, Himanshu Kaul, Amit Agarwal, Steven Hsu, Ram Krishnamurthy, Vivek De, and Sanu Mathew. 2019. A 1.4GHz 20.5Gbps GZIP decompression accelerator in 14nm CMOS featuring dual-path out-of-order speculative Huffman decoder and multi-write enabled register file array. In *2019 Symposium on VLSI Circuits*. C238–C239. https://doi.org/10.23919/VLSIC.2019.8777934

[54] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Hong Liu, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2016. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. *Commun. ACM* 59, 9 (aug 2016), 88–97. https://doi.org/10.1145/2975159

[55] Przemyslaw Skibinski. 2022. lzbench. https://github.com/inikep/lzbench.

[56] Akshitha Sriraman and Abhishek Dhanotia. 2020. Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, USA, 733–750. https://doi.org/10.1145/3373376.3378450

[57] Willy Tarreau and Dave Rodgman. 2018. LZO stream format as understood by Linux's LZO decompressor. https://www.kernel.org/doc/Documentation/lzo.txt.

[58] Kushagra Vaid. 2019. Hardware innovation for data growth challenges at cloud-scale. https://azure.microsoft.com/en-us/blog/hardware-innovation-for-data-growth-challenges-at-cloud-scale/.

[59] Kushagra Vaid. 2019. Improved cloud service performance through ASIC acceleration. https://azure.microsoft.com/en-us/blog/improved-cloud-service-performance-through-asic-acceleration/.

[60] Edward Wang, Colin Schmidt, Adam Izraelevitz, John Wright, Borivoje Nikolić, Elad Alon, and Jonathan Bachrach. 2020. A Methodology for Reusable Physical Design. In *2020 21st International Symposium on Quality Electronic Design (ISQED)*. 243–249. https://doi.org/10.1109/ISQED48828.2020.9136999

[61] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 609–621. https://doi.org/10.1145/3503222.3507731

[62] Wikichip. 2023. POWER9 - Microarchitectures - IBM. https://en.wikichip.org/wiki/ibm/microarchitectures/power9.

[63] Wikichip. 2023. Skylake (server) - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server).

[64] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. (May 2020).

[65] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343. https://doi.org/10.1109/TIT.1977.1055714