FlitReduce: Improving Memory Fabric Performance via End-to-End Network Packet Compression

Xingyu Li
University of California, Berkeley
xingyuli9961@berkeley.edu

Tushar Sondhi
University of California, Berkeley
sondhi.tushar@berkeley.edu

***Abstract*—New technologies in fabrication and packaging have led to an explosion in core counts as time continues. Network-on-Chip (NOC) is a router-based packet switching network that enables an efficient on-chip interconnect. Prior research has shown that, especially for highly parallel workloads, the design and performance of the NoC has a great effect on the overall performance of the system. Our project *FlitReduce* intends to improve the data transmission efficiency over the NoC by data compression. *FlitReduce* implements compressors and decompressors on the ends of the NoC to reduce the amount of overall memory traffic and, thus, the communication congestion, improving the NoC performance while imposing little effect on the other components in the network.**

The code of this project can be found on:
https://github.com/xingyuli9961/CS262AProject-FlitReduce-NoC_Compression.git

I. Introduction

Network-on-Chip (NoC) architectures are attract more attention as new fabrication and packaging technologies have led to an increase in core counts in the recent decade. This has raised the need for the increasingly scalable technologies to support the greedy need of more cores and larger caches. NoC architectures provide a unique intersection of scalability and performance that makes them an ideal target for such systems. However, the design and performance of the NoC is shown to have a great effect on the overall performance of the system [1]. This is because the NoC in these systems controls much of the communication that is occurring from the core to other parts of the system. In particular, this includes communication between local L1 cache on the core and L2 cache away from the core. There are some previous studies in the algorithms space or only simulated in a system-level software architecture simulator such as gem5 [2]–[6]. There are few actual hardware implementation designs tested in cycle-accurate manner [7]. We want to build actual hardware compressor designs with different compression algorithms, and evaluate the performance of message size and latency in the NoC system in Chipyard [8]. To do this, we will implement and end-to-end compression scheme over the NoC, which will compress and decompress data at the egress and ingress points into the network.

We will use SPEC2006 [9] and SPEC2017 [10] as representative workloads, which we have determined based on past literature [1]–[3], [7], [11], on a multi-core FireSim [12] simulation and produce memory traces from each of the cores. Generally, these are applications that take advantage of multiple cores making many accesses to memory. Since all cache-coherent communication is done over the NoC, it is very susceptible to memory congestion. These traces will contain the actual contents of the packet that is moving across the NoC. We can use these traces and use some simple software compression algorithms to profile several different compression techniques against packets produced by this representative workload. This will give us compression ratios that can represent a reduction in the number of packets moving through the system, while also showing memory congestion that can be used as a baseline performance metric.

The remainder of the paper is organized as follows. In Section II, we provide a brief overview on the NoC architecture, source of congestion, and data compression limitations. We perform a survey of compression algorithms in section III, including techniques used in prior literature and some algorithms proposed by ourselves. These compression strategies are then compared in software simulation against the aforementioned memory traces in Section IV. In Section V, we discuss the hardware implementation of our chosen compression techniques, while Section VI illustrates the experimental setup and results of the hardware evaluation. Several related works are compared in Section VII, before we discuss future works in section VIII and conclude the paper in Section IX.

II. Background

The underlying NoC architecture can be described as a combination of two components: a set of ”tiles” and the network interspersed between them. The ”tiles” in this system is any hardware block that needs to send information across the NoC. For our specific system, this includes CPU cores and L2 cache banks. In the systems currently available in the Chipyard infrastructure, multi-core systems with many cache banks are connected with a direct point-to-point crossbar, such that each core is directly connected to each L2 cache bank. The NoC replaces the crossbar with only direct interconnects between adjacent tiles. While this introduces a non-uniform memory architecture with multi-cycle latency just to transmit data between some tiles, it is seen as a more robust and scalable alternative to more traditional System-on-Chip (SoC) architectures [13].

The NoC architecture also introduces the concept of congestion. Packets are sent across the NoC in smaller segments

known as ”flits.” Due to limitations in the amount of information available, routing algorithms across the NoC are far from optimal and are susceptible to congestion, where multiple flits are competing for the same interconnect to a specific tile. This congestion can often result in performance degradation, if comparing identical systems with and without the NoC. This congestion becomes an issue especially in memory-intensive programs, as congestion increases along with the number of flits attempting to traverse the NoC at any given point of time. As such, congestion is a non-negligible weakness of NoC architectures.

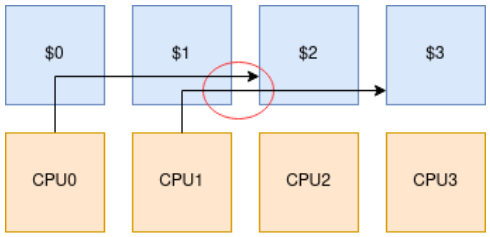

Fig. 1. Example of congestion from routing conflict

Any reduction in the number of flits also impacts congestion across the NoC, providing a potential pathway for performance improvement via end-to-end compression across the NoC. To further describe the context of the compression, it is important to know the bulk of the communication on the NoC is between the L2 cache banks and the L1 cache located within each CPU core. As such, the packets sent over the NoC are largely cache lines. Each flit sends a single 8B word of a 64B cache line, resulting in 8 flits sent across the NoC per cache line. The compression is done at a cache line granularity, and is only done on the payload (the 8B word) of the flit, and any control information that is sent alongside the payload for each flit is ignored.

In this context, the compression has 2 primary constraints. First of all, the granularity at which the compression is conducted is incredibly fine, meaning that not a lot of data is available to compress. This means that the compression is harder, as there are relatively fewer bits available in which patterns may arise that compression can target. Additionally, latency is a very important aspect of cache performance, and thus adding additional latency to the NoC can also result in performance degradation.

III. Algorithms

As discussed above, we intend to find efficient general-purpose compression algorithms that are optimized for 64-byte cacheline-size data and can be implemented as a small hardware block with feasible area utilization and time latency. We experimented and compared three different kinds of compression: Zero-encoding compression, NoΔ compression, and Frequent Pattern Compression. We also tried the mixed strategy that selects the best output from different algorithms.

A. Zero-Encoding Compression

It’s suggested by [6] that the data in the cache blocks contains a large amount of zero, and encoding the zeros in a more compacted pattern will reduce the amount of data that needs to be transmitted in the network. Our approach is to divide each flit evenly into smaller slots and use a bitmap header to represent if each slot is zero or not. In our design, 1 in the bitmap encodes that the value in this slot is zero, while 0 stands for a non-zero value. This might seem a bit counter-intuitive in the first glance, but it makes the checking process more convenient: if there are too few 1’s in the header, we decide not to compress the data as the algorithm will not reduce the data size. After compression, the original cache line data is reduced to a fixed-size header followed by only the non-zero values. There’s a trade-off between the finer granularity that encodes more zeros and a larger size of the bitmap header. We explore different levels of granularity, and more details will be discussed later.

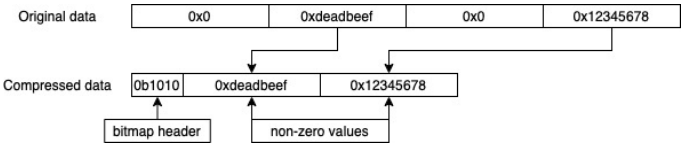

Fig. 2. Zero-Encoding Compression

B. NoΔ Compression

The original NoΔ compression algorithm is proposed by [2], which is inspired by the delta encoding [14], a method to transmit data in the form of differences rather than the complete version. NoΔ compression exploits the pattern that the data values stored in the same NoC packets may have a small range. In such case, we can use a common base value and an array of relative differences(Δs) to represent the original packet. Although the NoΔ compression sounds obvious, there are a lot of design options in how to configure the base and the differences. On one hand, we may set the base to be 32-bit or 64-bit, and we may have one or two base values for the entire cache. We may use the first value in the packet as the base, or we may dynamically calculate the base value according to the minimal and maximal values in the packet. On the other hand, we can have the Δs to be 1-byte or 2-bytes, and configured them to be signed or unsigned. All the choices will affect the portion of packets that are compressible and the size of the packet after compression.

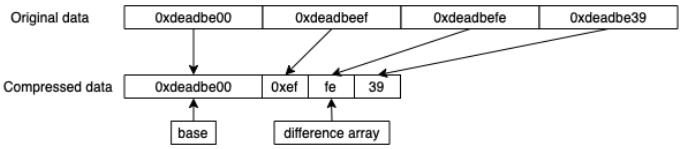

Fig. 3. Basic NoΔ Compression

We also propose our own modified version of the basic NoΔ compression, called ”Adaptive NoΔ compression”, that

allows partial compression. Adaptive NoΔ compression uses flit-size(64-bit) base and signed 1-byte Δs, and records an extra bitmap that indicates if a certain flit can be compressed or not. It selects the first flit of the original packet as the base and record 1 in the bitmap. For the rest of the 7 flits, it will record 1 in the bitmap and store the actual difference in the Δ array if it is within the compressible range; otherwise, it will record 0 in the bitmap, store 0 in the Δ array as space holder, and save the entire flit for later output. The compressed packet will have the base value as the first flit and the 1-byte bitmap and 7-byte Δ array as the second flit, followed by the uncompressed flits in the end.

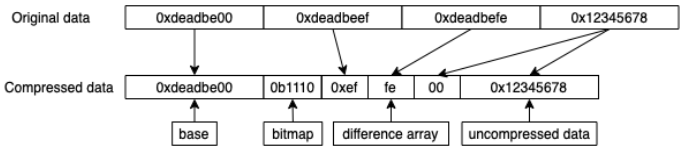

Fig. 4. Adapted NoΔ Compression

C. Frequent Pattern Compression

Frequent Pattern Compression [11] is a significance-based compression schemed designed for L2 caches, which has a slightly different use case than us. Significance-based compression suggests that a data value may be stored with fewer bits than the maximum bit allowed. Frequent Pattern Compression uses 32 bits as a word segment, and each word is compressed into a 3-bit header and the data of varied length, depending on its actual data value. For example, $0h07$ can be stored within 7 bits (4 bits plus a 3-bit header), which is much smaller than 32 bits. In our usage, we divide the entire packet into 16 32-bit segments, and follow the Frequent Pattern Compression to generated the decompressed packet. The detailed compression encoding is demonstrated in the table below for reference.

Prefix	Pattern Encoded	Data Size (bits)
000	Zero	0
001	4-bit sign-extended	4
010	1-byte sign-extended	8
011	2-byte sign-extended	16
100	2-byte padded with 0s	16
101	Two 1-byte sign-extended	16
110	Repeating bytes	8
111	Uncompressed word	32

TABLE I
FREQUENT PATTERN ENCODING

D. Mixed Compression Strategies

Although many papers focus on singular strategies, there is also the notion of using different types of compression in parallel, and choosing the best output. This is done in the NoΔ paper by using differently-sized bases or offsets [2]. However, few prior works explore the usage of different compression algorithms in parallel, and picking the best outcome for the compression technique. This technique leverages the advantage that different compression algorithms perform well on different sets of data, improving compression ratios beyond what is capable for any single type of compression, at the cost of increased area usage for the hardware.

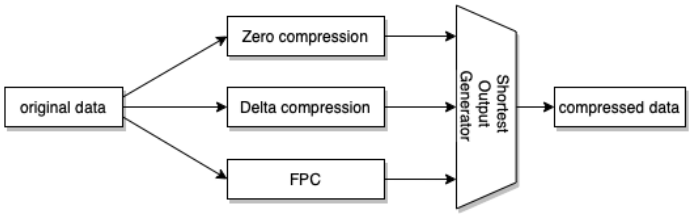

Fig. 5. Mixed Compression Strategy

IV. ALGORITHM EVALUATION

Directly implementing the algorithms in the NoC hardware takes excessive development and debugging time, and it is not suitable for comparing a large variety of compression strategies in an early stage. Therefore, in order to compare the performance of different algorithms in an agile manner, we developed a micro-benchmark that are representative for the SPEC memory access patterns. Running software simulation with the micro-benchmark takes much fewer time and significantly reduces the iteration time for hardware development.

A. MicroBenchmark

We intended to create a microbenchmark that reflects the actual data access pattern between the CPU cores and the caches. Initially, we wanted to use the latest version SPEC2017 to create the microbenchmark. However, there were no available binaries of SPEC2017 that could be run in the bare-metal mode. There was another option to run SPEC2017 in Qemu [15] emulator and output the memory trace, but there's no notion of cache in Qemu and writing our own L1 and L2 caches to analyzes the raw memory would took too much unnecessary work given the scope of the project. Thus, we selected four benchmarks (two floating point and two integer) in SPEC2006 [9] that can be run in bare-metal mode, including:

- soplex: it solves a linear program using a simplex algorithm and sparse linear algebra. Test cases include railroad planning and military airlift models.
- lbm: it implements the "Lattice-Boltzmann Method" to simulate in-compressible fluids in 3D.
- mcf: it uses a network simplex algorithm (which is also used in commercial products) to schedule public transport.
- sjeng: it runs a highly-ranked chess program that also plays several chess variants.

We ran the SPEC2006 program binaries in the Spike RISC-V ISA Simulator [16] with cache models to produce cacheline-granularity memory traces. We then randomly sampled 1000 contiguous logs per 100000 logs in the memory trace to generate the micro-benchmarks that would be used in the software simulations.

B. Software Simulation Result

We implemented the compressors of different algorithms in python and compared the compression ratio, which is defined as

$$compression\ ratio = \frac{No.\ flits\ in\ original\ data}{No.\ flits\ after\ compression}$$

Taking the NoC hardware limitations in mind, the smallest unit in the network is a flit, and we should compare the number of flits instead of the number of bits. For each algorithm, we profiled the flit compression ratios of sampled memory trace sets, and calculated the average. Overall, mcf and sjeng benchmarks tends to have smaller compression ratios, as our algorithm is more designed for the integer storage format. The best average compression ratio is 60.37% by the Mix Strategy. In other words, it can reduce roughly 40% traffic in the NoC Network, which is reasonably effective.

1) Zero-Encoding Compression: We compared four different configurations of the Zero-Encoding Compression algorithm, setting the granularity to 64bit, 32bit, 16bit, and 8bit, with a bitmap header of 8 bits, 16 bits, 32 bits, and 64 bits respectively. The best performance of the Zero-Encoding Compression demonstrates a compression ration of 76.52%, which reduces roughly $\frac{1}{4}$ of the flits. We observed that the compression ration decreased as we use finer granularity from 64bit to 16bit, but the performance becomes less effective when it moves to the 8bit. This is probably due to two reasons:

- Although finer granularity can compress more zeros, 8bit version requires a large 64-bit bitmap header, which takes an entire flit
- 16bit configuration requires a header of only 32 bits, and the first flits(64 bits) can fit in both the header and two non-zero values, which can reduce an extra flit in a lot of cases.

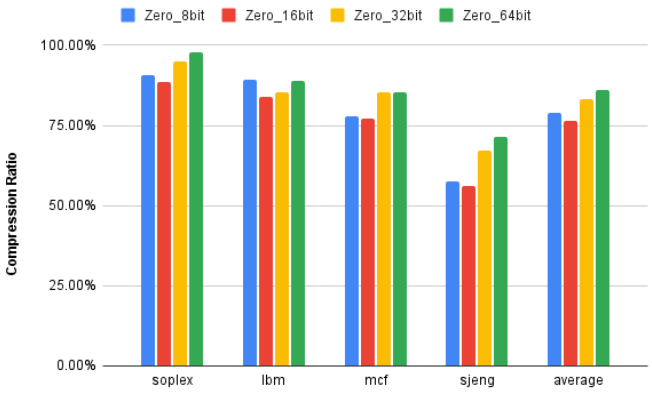

Fig. 6. Compression Ratios of Zero-Encoding Compression with different granularity

2) NoΔ Compression: We implemented 5 different NoΔ-family compression algorithms with 64-bit base values, including:

- 1-byte Δ with one common base value
- 1-byte Δ with two common base values
- 2-byte Δ with one common base value
- 2-byte Δ with two base values
- our Adapted NoΔ compression

We observed that NoΔ Compression had a generally higher compression rations than the Zero compression, and didn't perform well especially for benchmarks such as soplex and mcf. The best average compression ratio is 90.45% from the Adapted NoΔ compression. The results displayed in the diagram below also suggest that having two base values in a cacheline makes more packets compressible and reduces the number of flits, while the choice between 1-byte or 2-byte delta values doesn't have an meaningful effects. As expected, the Adapted NoΔ compression has the smallest compression ratio since it can compresses more packets as long as two other flits in the packet are within 1-byte the range from the first flit, while the original version requires all the flits to be in the range.

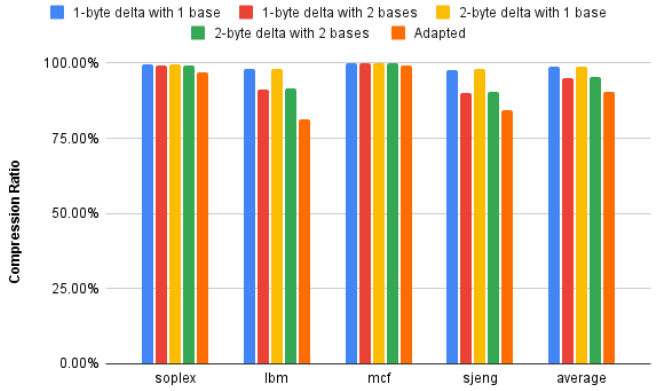

Fig. 7. Compression Ratios of NoΔ Compression of different configurations

3) Frequent Pattern Compression: The compression ratios of the Frequent Pattern Compression demonstrated its effectiveness over other algorithms, with an average compression ratio of 62.84%. The only drawback of the Frequent Pattern Compression is that the decompressed data has a lot of varied-length data segments, while Zero and Δ Compression only has arrays of fixed-size data. Such feature would make the combinational logic much more complicated in the hardware, and requires more development efforts in designing the actual physical data format.

4) Mixed Compression Strategies: We profiled two types of Mixed Compression Strategies: one is the combination of 16bit Zero compression and Adapted NoΔ compression, and the other is the combination of first strategy and the Frequent Pattern Compression. The results suggested that the mixture of Zero and NoΔ reached noticeable improvements for each algorithms, while the second mixed strategy showed smaller differences from the Frequent Pattern Compression alone.

5) Summary: According to the software simulation, Frequent Pattern Compression seems to be the most effective, though most complicated, algorithm, and the Zero-Encoding

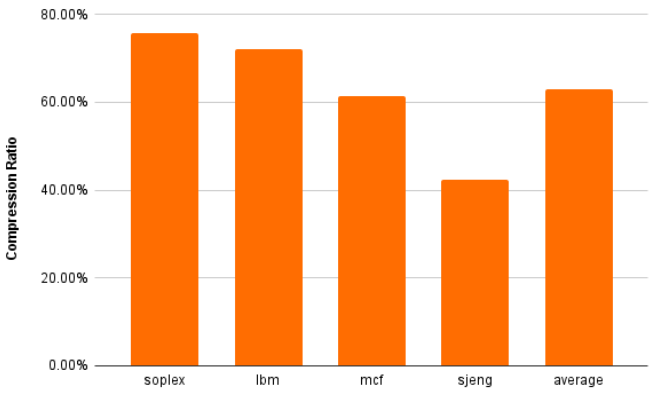

Fig. 8. Compression Ratios of Frequent Pattern Compression

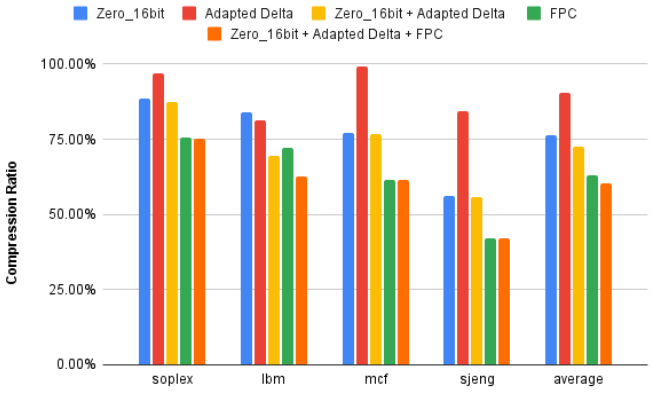

Fig. 9. Compression Ratios of Mixed Compression Strategies

Compression is better than the NoΔ Compression. Due to time limitation, we decided to implement the 16-bit Zero-Encoding Compression first, which has the best performance among the simple algorithms, and integrate the compressors and decompressors into NoC system for testing and profiling. It would be then easier to implement and compare other algorithm in the hardware, if we had already successfully set up the integration procedure.

Compression Algorithm	Average Compression Ratio
Zero 16bit	76.52%
Adapted Delta	90.45%
Mix(Zero 16bit + Adapted Delta)	72.39%
FPC	62.84%

TABLE II
AVERAGE COMPRESSION RATIO OF DIFFERENT ALGORITHMS

V. HARDWARE IMPLEMENTATION

In order to implement the compression scheme, we planned on building two hardware components that lie at each end of the network. Since read traffic usually makes up the majority of memory traffic, and is thus is particularly vulnerable to congestion, we focused on first implementing the compression scheme along the TileLink [17] D channel which handles data-bearing cache responses. While the compression system is relatively easy to implement across other data-bearing channels, this simplifies the initial implementation. Additionally, the compressor was built at the interface between the core and the NoC, and is thus just a simple drop in hardware block that can be enabled / disabled with ease.

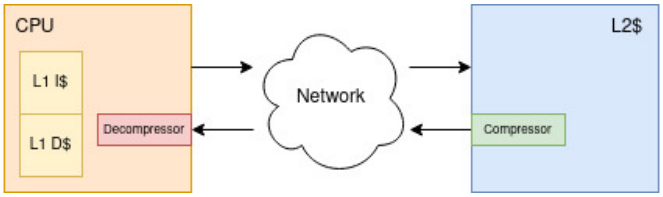

Fig. 10. Compression Scheme Hardware Placement

In the specification of the NoC that we are testing on, each flit is formatted with a lightweight header and a data payload, which contains both a single word of the cache and a lightweight TileLink [17] header for cache coherency purposes. Note that this is not the same as other NoC architectures, such as that used in FlitZip [5], which carries all of the header information in an entire flit at the head of the packet, which is why the FlitZip compression technique as incompatible with our system.

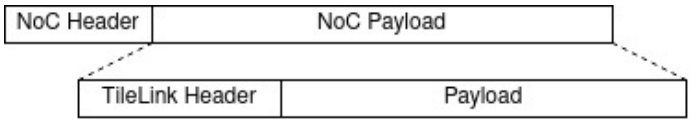

Fig. 11. NoC Flit Format

A. Compressor

Some limitations in the existing architecture leads to some performance inefficiencies in the compression scheme. The cache only outputs a single word at a time across the TileLink interface, meaning that the current implementation of the compressor has to wait 8 cycles for the entire cache line to enter the compressor. While this can be overcome by integrating the compressor deeper inside the cache, this greatly increases design complexity and thus was avoided for this iteration.

The compressor adds 8 cycles of latency, to every transaction, during which the input buffer is populated. For the following cycles, the compressor outputs the compressed version of the cache line. The compression is done entirely on the fly based on the bitmap generated from the values stored in the input buffer. As discussed in the prior section, the compressor implements 16-bit granularity Zero Encoding across the data in each flit, and all header information (both TileLink and NoC) is left untouched. The size of the Flit payload is increased by a single bit to accommodate a compressed bit which is used to indicate whether the data is compressed.

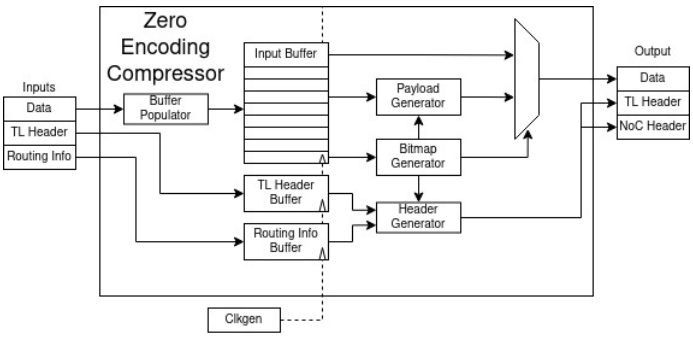

Fig. 12. Architecture of Zero-Encoding Compressor

B. Decompressor

The decompressor takes in the compressed flits and generates outputs on the fly, as long as there's enough information received. It has a latency of 2 cycles, as it is a safe design choice in the first iteration to meet all the timing requirements, and we may reduce its latency in future versions. There are an input buffer that stores all the unprocessed 16-bit non-zero data segments, and an output buffer that accepts the de-compressed flits when the out_enable is on.

The decompressor has three states: *idle*, *receiving*, and *running*. In the *idle* state, it either lets the uncompressed flits pass through or starts the decompression procedure when the head flit of a compressed packet is received. It records TileLink header information and the bitmap header for this flit, and transfers to *receiving*. In *receiving* state, it accepts all valid compressed flits and generates outputs simultaneously. Whenever a tail flit arrives, the decompression shifts to *running*, set the *in.ready* to *false* until the current packet is fully decompressed. Note that in the NoC we use, there is no re-ordering in the routers and all the flits should be received in the same order as it is sent. Hence, the decompressor would only receive valid uncompressed flits between packets, then it would only process uncompressed data in the *idle* state.

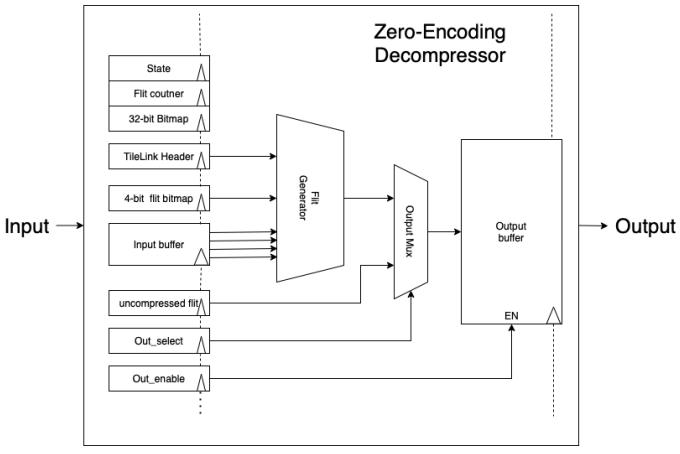

Fig. 13. Architecture of Zero-Encoding Decompressor

VI. Hardware Evaluation

A. Experimental Setup

In order to build and test the hardware, we took advantage of the open-source Firesim [12] and Chipyard [8] infrastructures. The advantage of this infrastructure is that it allows us to evaluate hardware designs on an FPGA via hardware emulation, which is a step above prior approaches which only attempt software implementations, using fpgas in cloud AWS instances via Firesim. Additionally, much of the framework we need to test NoC compression is already available as a part of Chipyard, making it a convenient platform on which we can build the compression system. We used the Constellation NoC generator to generate a 4 core, 16 cache bank arrangement described in the figure below. For the CPU cores, we are using Rocket in-order pipeline RISC-V cores that were developed here at UC Berkeley [18]. A more detailed description of the hardware used in the experimental setup is given in Table III.

Num. of Rocket [18] Cores	4
Num. of L2 Cache Banks	16
NoC Topology	2-D Torus
L2 Cache Size	512 KB
Num. of L2 Cache Banks	16
Num. of L2 Cache Sets	1024
Num. of L2 Cache Ways	8
L1 Cache Size	16 KB
Num. of L1 Cache Sets	64
Num. of L1 Cache Ways	4

TABLE III
HARDWARE CONFIGURATION OF EXPERIMENTAL SYSTEM

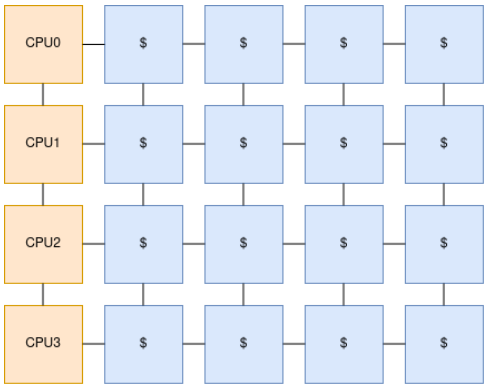

Fig. 14. Experimental NoC Setup

The amount of time it takes to run a benchmark like SPEC2017 in hardware emulation is extremely long (around 2-3 days to run the multicore intrate benchmarks), and can cost upwards of $ 500.00 in AWS credit fees. As such, we created a microbenchmark which approximates the memory accesses of the SPEC2006 [9] benchmark using the memory traces collected for the evaluation of the different compression algorithms. To do this, a file is created with the exact binary data as the memory trace. Then, the microbenchmark memory maps the file and reads the whole file multiple times. In this way, the microbenchmark induces read traffic in which the

data matches the accesses of the actual benchmark run, while running for a much shorter period of time.

B. Results

1) Area Utilization: The area utilization of the compression system is relatively small, although not negligible. As seen in Table III, the overall utilization of the design as a whole increases minimally, as it only grows by around 8%. However, the compression makes up a non-negligible area on the NoC, which increases in size by 58% with the compression system added. This measurement was taken with a 2 core, 2 cache system, in which there are 2 compressors and 2 decompressors. This smaller, dual core design was used due to timing constraints, as the larger quad core design can take upwards of 12 hours to build.

Component	Percentage Area Increase
Total	7.91%
NoC	58.24%

TABLE IV
AREA UTILIZATION FROM COMPRESSION SYSTEM

2) Approximate MicroBenchmark: The MicroBenchmark was run on both the NoC architecture described earlier and an identical system running Quad Core Rocket with a point-to-point XBar system. This was conducted to show how the NoC introduces performance degradation compared to an otherwise identical network. The total runtime of the MicroBenchmark increases by 15% with the addition of the NoC, which can be attributed to congestion in the system. With the addition of the Zero-Encoding Compression system, which reduces the congestion to around 75% on average, this performance degradation can be decreased to around 11%. While we would also like to run the benchmark on the final system with the compression implemented on hardware, unfortunately some edge cases in the memory system prevents us from doing so at this moment.

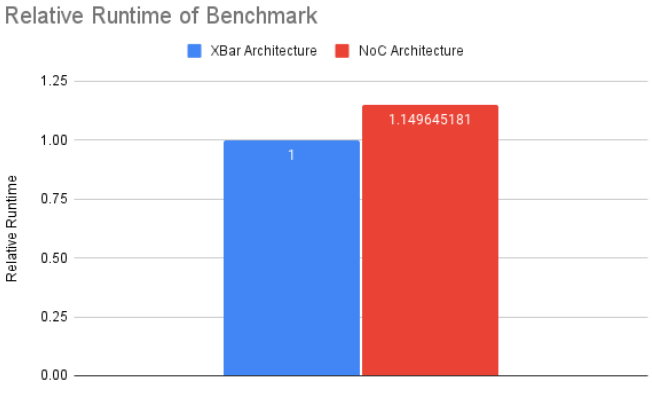

Fig. 15. Approximate MicroBenchmark on XBar and NoC systems

3) Full SPEC2017 Benchmark: Scores in this section are relative to a baseline system given by a baseline system. [1]. Eventually, we plan to have full SPEC benchmark runs with the NoC system implemented, to in order to look at total end-to-end performance improvements due to compression across the NoC.

SPEC2017 MicroBenchmark	SPEC Score
exchange2	4.314
leela	2.062
deepsjeng	0.985
x264	1.416
perlbench	0.992
gcc	0.901
omnetpp	0.709
mcf	0.740

TABLE V
SPEC2017 SCORE ON QUADCORE ROCKET WITH NOC CONFIG

VII. RELATED WORK

A lot of previous works discussed about the compression in the cache, such as [19], which provides a brief survey of the proposed algorithms and challenges in cache compression, and [11] that proposes the Frequent Pattern Compression designed for the L2 Caches. More recent papers take the NoC architecture into account in the discussion of data compression: [7] discussed compressors in both the cache and the NIC before the data is inserted into the network, [20] uses idle CPU resources to compress data in software before sending out, and [4] proposed the adapted DISCO data compressor [3] with an scheduler in routers to decide when and which packets to be compressed.

[5] observes that existing delta compression in NoC cannot reduce data redundancy in packet effectively and simultaneous computation are costly in the current design. It proposes FlitZip Compression that has more dynamic compression tactics, which make more packets compressible. However, to do so, FlitZip takes advantage of system-specific specifications, namely the smaller packet size and much larger header flit which has space to store a specialized bitmap per packet. This makes this compression technique infeasible outside a small subset of NoC systems, which does not include ours.

Mentioned in the Algorithms Section before, [6] proposes Zero-Encoding Compression and [2] proposes NoΔ compression. Both of them have more similar NoC constraints as ours, but they only test their design in the gem5 simulator [21] without any real RTL design implementations.

VIII. FUTURE WORK

There are several areas in which we can improve the work in the current paper. For one, finalizing the implementation of the Zero Encoding Compressor in RTL and getting real programs to run on the NoC in hardware emulation is of utmost priority, as it will bring to light the performance of the current system with this type of compression. Afterwards,

[1] https://www.spec.org/cpu2017/results/res2017q2/cpu2017-20161026-00003.html

it would be useful to try several more complex compression techniques, such as the Mixed Compression, as that was shown to have the best compression ratios in our survey of various compression algorithms.

On attempting to implement the Zero-Encoding Compression, we noticed several drawbacks to compressors that were ill-explored in other works. For example, few of the other papers describe latency-optimizations that are often made in more complex cache systems. For example, it is possible that there is a critical word first policy between the L1 and L2 caches. Having a compression scheme, which by default has to add some amount of latency, is dangerous for latency-sensitive applications.

While some works, such as DISCO [3], have explored schedulers in the routing to try and determine what should be compressed, we believe there may be an interesting cross-section between compression work and Quality-of-Service. In QoS systems, data that is traveling around the chip is separated into different groupings which have varying QoS requirements [22]. As such, packets with few QoS requirements can be selectively compressed, allowing for more bandwidth to be allocated to more latency-sensitive packets, while effectively maintaining the same bandwidth for low-priority packets.

The idea for compressing low-priority packets can also be applied in systems without QoS. For example, any system that takes advantage of prefetching in the cache can use compression on prefetched packets with no downside [23]. This is because prefetched packets are by definition latency-insensitive, and often a weakness cited by cache prefetchers is the fact that they can often hog memory resources needed by more sensitive applications [23]. Compression is a simple technique that can be implemented to alleviate this issue, and perhaps even lead to more aggressive prefetching policies due to the extra bandwidth potentially provided by compression.

In terms of purely the hardware design, the added latency of the design could likely be improved. In particular, due to some restrictions in the way we have implemented the compressor, it receives as single word from the cache as an input at a time. Theoretically, it is possible to implement a cache-line-lookahead methodology that is able to draw out an entire cache line at a time, reducing the 8-cycle latency to only a single cycle to produce the bitmap.

IX. Conclusion

In this paper, we introduced *FlitReduce*, an NoC performance augmentation strategy that reduces communication congestion via data compression. We first explored different feasible data compression algorithms within the NoC platform constraints. We then developed a SPEC-memory-access-pattern-representative microbenchmark and analyzed the effectiveness in flit reduction of different algorithms in software simulation. In consideration of the compression ratio and complexity, we selected Zero-Encoding Compression with 16bit granularity for the hardware implementation.

A successful hardware implementation of the compression algorithm set should possess following features: correctness, high flit compression rates, low area utilization, and low latency. In the bare-metal model, the decompressor successfully generates the same output flit as the input of the compressor, and reduces more than 20% of flits in transition in unit tests. The area utilization is relatively small and acceptable, which only increase less than 8% percent area in a Dual-core Dual-Cache chip. However, when we integrated the system and attempted to run some larger benchmarks, some disturbance occurred and we didn't manage to fix all the bugs given the time limit. In terms of latency, our design adds 8 cycles with the compressor and 2 cycles with the decompressor, totalling in 10 total cycles of added latency. However, this choice was an artifact of the design decision to implement the compression system entirely as a part of the Constellation NoC generator. Additionally, from a design standpoint, attempting to integrate the compression / decompression system across different repositories for the L1 and L2 cache seemed incredibly difficult due to the existing implementations of these hardware systems.

Acknowledgment

We would like to thank Professor John Kubiatowicz and GSI Stephanie Wang for their continued support throughout the semester. Additionally, several members of the Adept Lab, namely Sagar Karandikar and Jerry Zhao, have provided guidance at every step of the way, and played an important role in the benchmark development and compression implementation development. Finally, we would like to thank developers and retainers of development tools such as Firesim and Chipyard, as well as hardware generators such as Rocket Chip and Constellation (an in-progress NoC generator by Jerry Zhao) which form the foundation with which this project was built upon.

References

[1] H. Cota de Freitas, L. M. Schnorr, M. A. Z. Alves and P. O. A. Navaux, "Impact of Parallel Workloads on NoC Architecture Design," 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, 2010, pp. 551-555, doi: 10.1109/PDP.2010.53.

[2] J. Zhan, M. Poremba, Y. Xu and Y. Xie, "No Δ : Leveraging delta compression for end-to-end memory access in NoC based multicores," 2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC), 2014, pp. 586-591, doi: 10.1109/ASPDAC.2014.6742954.

[3] Y. Wang, Y. Han, J. Zhou, H. Li and X. Li, "DISCO: A low overhead in-network data compressor for energy-efficient chip multi-processors," 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), 2016, pp. 1-6, doi: 10.1145/2897937.2898007.

[4] Y. Wang, H. Li, Y. Han and X. Li, "A Low Overhead In-Network Data Compressor for the Memory Hierarchy of Chip Multiprocessors," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 37, no. 6, pp. 1265-1277, June 2018, doi: 10.1109/TCAD.2017.2729404.

[5] D. Deb, R. M.K. and J. Jose, "FlitZip: Effective Packet Compression for NoC in MultiProcessor System-on-Chip," in IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 1, pp. 117-128, 1 Jan. 2022, doi: 10.1109/TPDS.2021.3090315.

[6] K. Rani and H. K. Kapoor, "ZENCO: Zero-bytes based ENCOding for Non-Volatile Buffers in On-Chip Interconnects," 2020 57th ACM/IEEE Design Automation Conference (DAC), 2020, pp. 1-6, doi: 10.1109/DAC18072.2020.9218620.

[7] R. Das et al., "Performance and power optimization through data compression in Network-on-Chip architectures," 2008 IEEE 14th International Symposium on High Performance Computer Architecture, 2008, pp. 215-225, doi: 10.1109/HPCA.2008.4658641.

[8] A. Amid et al., "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs," in IEEE Micro, vol. 40, no. 4, pp. 10-21, 1 July-Aug. 2020, doi: 10.1109/MM.2020.2996616.

[9] J. L. Henning, "SPEC CPU2006 benchmark descriptions," ACM SIGARCH Computer Architecture News, vol. 34, no. 4, pp. 1–17, 2006.

[10] J. Bucek, K. Lange, and J. v. Kistowski. 2018. "SPEC CPU2017: Next-Generation Compute Benchmark". In Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18). Association for Computing Machinery, New York, NY, USA, 41–42. DOI:https://doi.org/10.1145/3185768.3185771

[11] A. R. Alameldeen and D. A. Wood. "Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches". Technical Report 1500, Computer Sciences Department, University of Wisconsin-Madison,April2004.

[12] S. Karandikar et al., "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud," 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), 2018, pp. 29-42, doi: 10.1109/ISCA.2018.00014.

[13] L. Benini, D. Bertozzi. "Network-on-chip architectures and design methods", in Computers and Digital Techniques, IEE Proceedings -. 152. 261 - 272. 10.1049/ip-cdt:20045100. (2005).

[14] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. 1997. "Potential benefits of delta encoding and data compression for HTTP". SIGCOMM Comput. Commun. Rev. 27, 4 (Oct. 1997), 181–194. DOI:https://doi.org/10.1145/263109.263162

[15] F. Bellard. "QEMU, a Fast and Portable Dynamic Translator." USENIX Annual Technical Conference, FREENIX Track (2005).

[16] Spike RISC-V ISA Simulator, GitHub repository, https://github.com/riscv-software-src/riscv-isa-sim.git

[17] H. Cook, W. Terpstra, and Y. Lee, "Diplomatic design patterns: A TileLink case study," in Proc. 1st Workshop Comput. Archit. Res. RISC-V, 2017.

[18] K. Asanović, A. Rimas, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, P. Dabbelt, J. R. Hauser, A. M. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moretó, A. J. Ou, D. A. Patterson, B. C. Richards, C. Schmidt, S. Twigg, H. D. Vo and A. Waterman. "The Rocket Chip Generator." (2016).

[19] D. R. Carvalho, A. Seznec. 2021. "Understanding Cache Compression". ACM Trans. Archit. Code Optim. 18, 3, Article 36 (June 2021), 27 pages. https://doi.org/10.1145/3457207

[20] B. Welton, D. Kimpe, J. Cope, C. M. Patrick, K. Iskra and R. Ross, "Improving I/O Forwarding Throughput with Data Compression," 2011 IEEE International Conference on Cluster Computing, 2011, pp. 438-445, doi: 10.1109/CLUSTER.2011.80.

[21] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and Da. A. Wood. 2011. "The gem5 simulator". SIGARCH Comput. Archit. News 39, 2 (May 2011), 1–7. DOI:https://doi.org/10.1145/2024716.2024718

[22] F. Feliciian and S. B. Furber, "An asynchronous on-chip network router with quality-of-service (QoS) support," IEEE International SOC Conference, 2004. Proceedings., 2004, pp. 274-277, doi: 10.1109/SOCC.2004.1362432.

[23] J. Tse and A. J. Smith, "CPU cache prefetching: Timing evaluation of hardware implementations," in IEEE Transactions on Computers, vol. 47, no. 5, pp. 509-526, May 1998, doi: 10.1109/12.677225.