Cacheline Utilization-Aware Link Traffic Compression for Modular GPUs

Kishore Punniyamurthy
The University of Texas at Austin
kishore.punniyamurthy@utexas.edu

Shomit Das
AMD Research
shomit.das@amd.com

Andreas Gerstlauer
The University of Texas at Austin
gerstl@utexas.edu

***Abstract*—Modular GPU systems are increasingly gaining attention for continued scaling of system sizes under packaging and fabrication challenges. In such systems, limited interconnect bandwidth can be a concern for system performance and energy efficiency. Compression has been widely considered as an effective means to reduce the amount of data moved. However, most prior related schemes are limited to exploiting the cacheline data values for achieving high compression ratios.**

We propose CUALiT: Cacheline Utilization-Aware Link Traffic compression to reduce I/O link traffic for modular GPU systems. Our approach exploits the variation in temporal and spatial utilization of individual cacheline words to achieve higher compression ratios. We utilize a novel mechanism to predict utilization of cachelines across warps at word granularity. Predicted unutilized words are dropped from responses. Furthermore, latency-critical words are compressed using traditional methods while words with temporal slack are coalesced across cachelines and compressed lazily to achieve higher compression ratios. CUALiT reduces offchip link traffic by 14% on average while achieving up to 25% lower system energy and an average 11% (up to 2x) higher performance over a compression only scheme.

I. Introduction

System performance can be significantly impacted by interconnect bandwidth constraints in current hardware designs [1]. With a growing number of scalable systems utilizing highly parallel processors [2]–[4], increasing fabrication challenges of large chips, and higher data movement costs [5], interconnect bandwidth has become a very valuable resource.

Reducing data traffic is a widely recognized way to save interconnect bandwidth, thereby improving performance and reducing memory energy consumption. Compression is an established solution for reducing the amount of data moved between memories and processors. While many different compression approaches have been proposed in the past [6]–[8], they typically only exploit redundancy in data values for compression but otherwise treat all words (4 bytes) in a cacheline identically. At the same time, general purpose graphics processing unit (GPGPU) applications exhibit significant variation in temporal and spatial cacheline utilization [9], [10]. On the one hand, there are words that are accessed either immediately after the cacheline is brought in, or have some timing slack before they are first accessed (temporal utilization). On the other hand, there are words that are never accessed during their lifetime in the cache (spatial utilization). Such variations can be exploited to further reduce the amount of data moved. Adaptive cache management schemes that predict spatial locality of memory requests and support partial cacheline accesses [10], [11] to reduce bandwidth consumption have been proposed. However, these schemes are limited

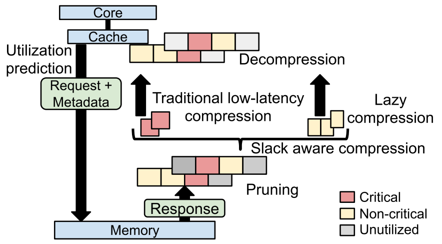


Fig. 1: CUALiT Overview.

in their ability to perform word-granularity prediction for GPGPU applications and thus reap lower benefits. LATTE-CC [12] relies on parallelism in GPUs to perform aggressive but slower compression. However, it is applied per individual cacheline and does not capture opportunities across cachelines.

In this paper, we present CUALiT (pronounced "quality"), an approach for Cacheline Utilization-Aware Link Traffic compression that addresses such concerns by exploiting both spatial and temporal variation in cacheline utilization at word-level granularity to reduce interconnect traffic. An overview of our scheme is shown in Fig. 1. It uses a novel mechanism for fine-grained prediction of cacheline utilization. The words predicted to be unutilized during the lifetime of a cacheline are dropped from memory responses. Words with predicted slack in access times are batched across cachelines and lazily compressed. This enables us to harness data content locality [13] across different cachelines being accessed at the same time, resulting in higher effective compression ratios and subsequent traffic reduction. The critical words are handled in similar manner as in traditional compression schemes.

In summary, this paper makes the following contributions:

- We analyze the variation in spatial and temporal utilization across cacheline words for GPGPU applications.
- We propose a novel prediction mechanism that allows fine-grained prediction of cacheline utilization for GPGPU applications.
- We propose lazy batch compression, an effective mechanism to exploit the temporal slack across words within a cacheline to achieve higher compression ratios.

The rest of the paper is organized as follows: we show the opportunity for exploiting cacheline utilization to reduce bandwidth consumption in Section II followed by an overview of related work in Section III. Section IV provides a detailed description of our scheme. Our experimental setup and results are presented in Section V. Finally, the paper concludes with a summary in Section VI.

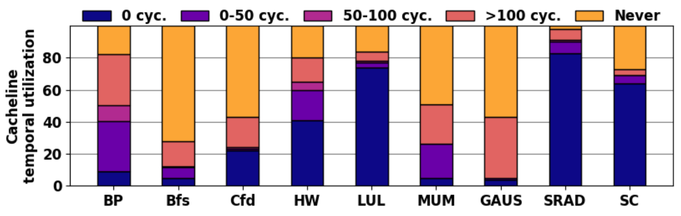


Fig. 2: Cacheline utilization.

II. Motivation

Fig. 2 shows the distribution of word utilization in cachelines across different GPGPU benchmarks. We track the time of first access to each word relative to the time the cacheline was brought into the cache. Not surprisingly, we can observe that the different words within a cacheline are requested at different times. Based on utilization, they can be categorized into three groups: *Critical words*: which are accessed immediately after cacheline becomes active in cache (31% on average). *Non-critical words*: which have a delay between cacheline filling and being accessed. We see that $\sim$10% of words are accessed within 50 cycles of filling cacheline and $\sim$17% of words have a slack of more than 100 cycles. *Unutilized words*: which are never accessed before the line gets evicted ($\sim$40%). In conclusion, we see that cachelines exhibit variation in both spatial and temporal utilization. Such variations can be leveraged to reduce the data movement. Spatial variation can be used to prune the unutilized words in memory responses while the slack available due to temporal variation can be used to increase data compression opportunities. However, there are challenges in doing so. 1) *Accurate utilization prediction*: accurate prediction of utilization is vital since an aggressive incorrect prediction will result in additional accesses and performance loss while conservative prediction will limit benefits. Prior works have proposed spatial utilization prediction but they provide coarse granularity prediction [10] or are intended for CPU applications [11] and may not work as well for GPU applications, as explained later. We address this by introducing a novel granularity predictor which can predict utilization and prune the unutilized words from memory responses. 2) *Exploiting slack to reduce data movement with minimum overhead*: traditional compression schemes aim to compress cachelines under stringent latency constraints (latency-sensitive). Conversely, exploiting temporal variation requires schemes which optimize for compression ratio with long compression/decompression latency and low hardware overhead. We propose a lazy batch compression scheme which delays responses in order to find and harness data content locality across cachelines and thereby achieving higher compression ratio by exploiting the slack.

III. Related Work

Researchers have proposed many compression schemes focusing on minimizing decompression latency for cache and link compression. Most of the earlier proposed works tend to focus on removal of redundant data patterns in cacheline data. BΔI [6] is based on the observation that most cachelines have a low dynamic range for data and hence, can be represented as base and series of offsets using fewer bytes than actual cacheline size. There have been works which exploit frequent values [7] or frequent patterns within cacheline [13], [14]. Some schemes have suggested a data transformation step that results in data with higher compression ratio [15]. CUALiT is orthogonal to all these compression algorithms since it uses existing schemes to perform compression while considering the spatial and temporal utilization. We evaluate CUALiT using the C-PACK [14] compression scheme; however, any scheme which compresses across lines can be used instead. LATTE-CC [12] aims at aggressively compressing individual cachelines by switching between slow but effective and fast but relaxed compression schemes depending on the parallelism available in GPUs. CUALiT aims at reducing link bandwidth consumption (and not cache compression) by exploiting the slack in access times available across cachelines. Further, aggressive single cacheline compression schemes used in LATTE-CC can equally be used in CUALiT.

There have been prior works which have proposed partial cache line access to reduce link bandwidth usage and potentially increase effective cache size. LAMAR [10] proposes dynamically switching between performing 32B cacheline sector accesses against entire 128B cacheline access depending on the prediction mechanism used. Further, they also suggest modifications to caches to increase their effective size. CUALiT does aim to reduce bandwidth requirements depending on the spatial locality captured (cacheline utilization) similar to these approaches, but we perform much finer granularity accesses (as small as 4B). Such small granularity accesses make predicting the number of words to be accessed and which words to be accessed more challenging. We do not propose changing effective cache size but such modifications can very well be performed. *Amoeba-cache* [11] proposes variable cacheline size to reduce link bandwidth consumption and cache capacity utilization. This work also proposes a prediction mechanism to perform fine grained access inspired from earlier works [10]. The mechanisms presented in these works are intended for CPUs and can perform poorly for GPUs (shown later). CUALiT uses a novel prediction mechanism to predict the correct granularity of access and we show that it avoids the pitfalls of prior works.

IV. CUALiT

CUALiT combines granularity prediction, utilization-aware pruning, and compression to reduce the amount of data moved across the interconnect; thereby saving bandwidth and energy consumption while improving performance. Fig. 3 shows the overview of our scheme. Our scheme begins with an accurate prediction of the utilization of the requested cacheline during its lifetime in the cache. To do so, the prediction mechanism needs to be trained. Caches are augmented with book-keeping data to track the utilized words within the cacheline. Every time a cacheline gets evicted, its utilization is used to calibrate the granularity predictor ❶ (more details in Section IV). When a memory read request is generated due to a cache miss, the utilization of the cacheline is predicted and then the request along with this metadata is sent across the interconnect to the

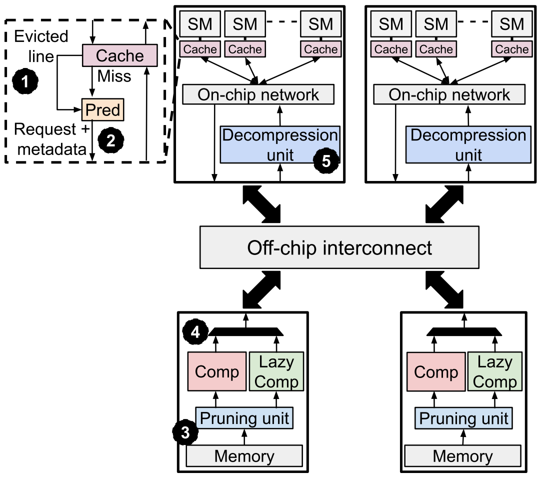


Fig. 3: CUALiT Block Diagram.

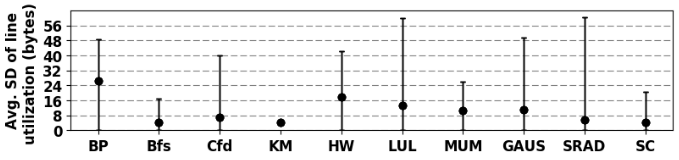
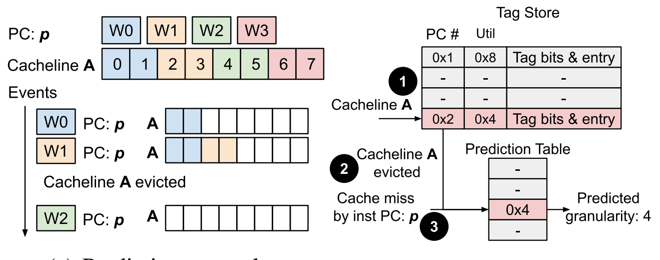


Fig. 4: Access granularity variation.



(a) Prediction example.
(b) Granularity predictor

Fig. 5: Utilization prediction.

memory ❷. Once the responses have been accessed from the memory, they are pruned ❸ and compressed based on their criticality ❹ before sending into the interconnect. Finally, the compressed responses are decompressed within each GPU module before being filled into the cache ❺. The metadata included with the responses specify the valid words within each response. We now delve into the individual components.

A. *Utilization prediction*

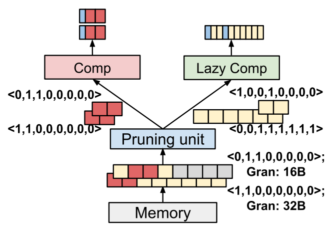
Benefit from CUALiT hinges on reliable and fine-grained prediction of cacheline utilization. In a GPU, since the threads are mostly identical, it is reasonable to expect the same instructions across warps to behave similarly. We validate this by tracking the variation in the utilization of cachelines accessed by individual instructions (program counter (PC) specific) and plot the standard deviation (SD) of utilization (in bytes) averaged across all PC for different benchmarks as shown in Fig. 4. The maximum and minimum SD observed are plotted as error bars. We see that for most benchmarks, the variation in utilization of cachelines accessed by same PC is $\sim$10B or less for a 128B cacheline, indicating that instructions with same PC have similar utilization in most cases. We exploit this observation in our scheme.

Prior works have proposed fine-grained utilization prediction by selecting useful words as the set of all words between touched words seen in history, while utilizing critical word index [11]. Such schemes may not work well for GPU applications. Consider the example shown in Fig. 5a consisting of 4 warps (W_0-W_3) each accessing 2 words in cacheline A (32B cacheline) on executing instruction with program counter (PC) p. Consider a case where cacheline A is filled in to the cache and gets evicted by the time only 2 warps (W_0-W_1) have completed their accesses. Prior PC-based spatial prediction solutions [11] use the utilized words to predict the mask for next access, which is <0:3>in this case. Next consider that warp W_2 executes PC p, now the prediction will be either <0:3>in which case the prediction is wrong, or if the predictor uses critical word index then no prediction will be made and result in conservative access. Similar miss predictions may happen when warps access other cachelines while executing PC p instruction.

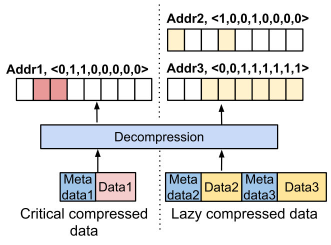
We propose a novel prediction mechanism capable of reliably predicting the number of words (4B) which will be accessed from the requested cacheline before its eviction. The predictor outputs the granularity for the access and not the pattern of useful bytes within cacheline. Consider accessing 128B cacheline at different granularities for example, 1x128B, 2x64B, 4x32B and so on forming a hierarchy. The predictor predicts the optimal granularity (4/8/16/32/64/128 bytes) and only the predicted granularity size segments containing the requested words are moved. For example, if the coalesced request for the cacheline requires the 0^{th} and 5^{th} word and the predicted granularity is 16B. Then all the words within the 16B segments containing requested words (i.e., words 0-7) are brought in response. For the example shown earlier, our prediction mechanism will learn the granularity to be 4 words and would predict the 4 word chunk containing the requested words (i.e., <4:7>) when W_2 executes.

Fig. 5b shows the working of the granularity predictor. It is a PC-based predictor consisting of a prediction table and metadata tracking as part of the cache tag store. The prediction table is indexed by PC bits and holds the predicted granularity for the PC. The tag store is extended to store bits from PC making the request and to track the cacheline words accessed.

Predictor training is a two step process. Every time a new cacheline is allocated ❶, bits from PC of the corresponding instruction are stored as part of tag entry. We use bits <10:7>from PC. On a line eviction ❷, the PC bits and utilization metadata are used to update the prediction table. The PC bits are used to index into the table and the corresponding entry is updated with the new granularity (4/8/16/32/64/128 bytes) depending on the number of words accessed in the evicted cacheline. Since the update to the predictor table happens on eviction, it does not affect the critical path and therefore, does not add to the memory latency. We found that predictor table with 16 entries worked well for our experiments. Finally on a cache miss ❸, bits from the requesting PC are used to index into the granularity prediction table to obtain the granularity of request. If no prediction is



(a) Pruning and compression.



(b) Decompression.

Fig. 6: CUALiT working.

available, full cacheline is selected by default. The predicted granularity along with the bytes required by the coalesced request is used to determine the actual utilization of the cacheline. The total overhead of metadata sent along with each request is 35 bits (32b bitmask + 3b for granularity).

The above scheme predicts the spatial utilization of cachelines and not its temporal utilization. We mark the words required in the coalesced request as critical and others as non-critical. We do so for simplicity as tracking variation in slack across words will require a complicated design and additional book-keeping. As results (in a later section) show, this simple differentiation is sufficient to get benefits.

B. Spatial utilization-aware pruning

The predicted spatial utilization is used to prune the unutilized words from the responses. This is simply done by dropping the words from the memory responses and updating the metadata to indicate the valid words. Fig. 6a shows an example of the same with two 32B responses with predicted granularity of 16B and 32B respectively. The unutilized words (shown in gray) from memory responses are cropped by the pruning unit. Instead of pruning the memory response, it is possible to read only the required bytes, resulting in potential lower memory access latency. However, in our experiments we read the entire cacheline from memory. Unlike spatial utilization, exploiting temporal utilization is not trivial since all the words are required but have different slack time. Traditional compression schemes optimize for latency and treat all words equally except for their data content. On the contrary, we need schemes which optimize for compression ratio at the expense of long latency.

C. Slack-aware compression

We perform slack-aware compression where words are compressed differently based on the available slack (criticality). As mentioned earlier, words are marked critical if they are required by the request causing the cache miss (using metadata in each request). In our example (Fig. 6a), words <0:3>of the first response are predicted to be utilized based on its granularity with words <1:2>as critical while the second response has entire cacheline predicted to be utilized with words <0:1>as critical. Critical words which are required immediately are compressed using the traditional scheme by the *Comp* unit. We use C-PACK as our standard compression scheme but any other scheme can equally be used. The non-critical (and any mispredicted critical) words are batched together and compressed lazily. Each compressed data also

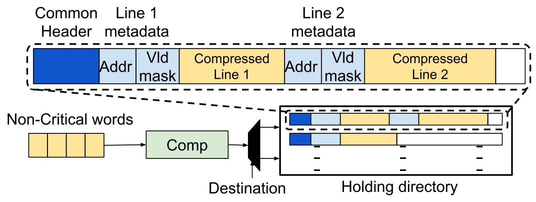


Fig. 7: Lazy batch compression.

contains metadata (a bit mask) indicating the number and position of valid words.

Fig. 7 shows the *lazy batch* compression unit in detail. Lazy batch compression tries to compress non-critical words across cachelines with an aim to achieve higher compression ratio. It uses a library based compression scheme (C-PACK) to compress words while reusing dictionary across cachelines to exploit data-content locality and thereby achieving higher compression ratio. To be able to compress across cachelines, the responses are delayed and batched together by storing them in the *holding* buffer for a certain duration (a parameter). Delaying and batching responses to compress more words allows this scheme to trade-off slack for compression ratio. In a multi-module GPU system memory responses might be destined to different modules and should be batched separately. We do so, by having a *holding* directory with a buffer for each destination. The buffer size is set at 256B to limit overhead. When a new response with slack reaches the unit, the appropriate buffer is selected based on the destination, the incoming words are compressed using the destination-specific dictionary and inserted into the buffer if space available. For every new cacheline inserted, its valid mask (32b) and address (48b) are also stored. The responses are dispatched to their destinations once the delay duration is over or the buffer is full. If there is insufficient space in buffer, the words are treated the same as critical words and dispatched immediately without delay. We use 32B dictionary per destination and delay duration of 200 cycles to keep the overhead nominal.

D. Decompression

The compressed responses received from the interconnect are decompressed at each module before filling them into caches. Since lazy batch compression compresses lines to same module (not same cache), decompression is done at the I/O interface of each module as shown in Fig. 3. The metadata included with every response received contains information about the number and position of valid words. Fig. 6b shows an example of decompression in action. *Metadata1* provides the address (*Addr1*) and valid mask (<0,1,1,0,0,0,0>) for critical compressed line which is used for its decompression. Lazy batch compressed responses with multiple cachelines are decompressed one at a time using the metadata pertaining to individual cacheline. The decompression mechanism is same as the original scheme [14] except the number of words are determined by valid mask. The extracted responses are then sent to their respective caches within each module where they are filled into the cache along with their active word masks which are stored in tag arrays. We add compression/decompression latency overhead of 10 cycles [14]. If a read request wants a

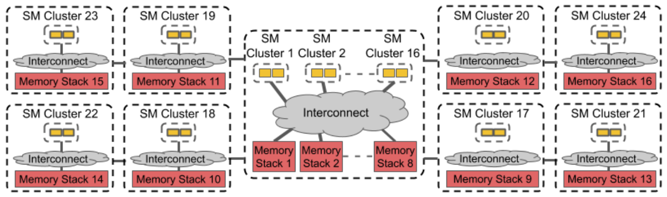


Fig. 8: Example of Simulated System.

TABLE I: Simulation parameters.

System Configuration	
Total SMs Memory stacks	64 (16 clusters with 4 each) 16
SM Configuration	
Core configuration	1.4Ghz, GTO warp scheduler [16]
Private L1 cache	32kB, 4-way, 128B cacheline [20]
Shared L2 cache	1MB, 16-way, 128B cacheline [20]
Interconnect	
Frequency	2.5Ghz
Topology	Ext: Tree, Central pkg: Fully connected
Bandwidth	
In-pkg bandwidth	160 GB/s per stack [20]
Off-chip bandwidth	80 GB/s [20]
Memory Stack	
Stack configuration	16 vaults/stack, 64 TSVs/vault [20]
Scheduling policy	FR-FCFS
Timing	DDR3 [20]

word which is not active in the cache, it is treated as a miss. Since this work aims at reducing interconnect traffic, caches in our experimental setup do not support variable cacheline size. Therefore, the effective cache capacity remains the same.

V. Evaluation

In this work, our goal is to reduce the link bandwidth consumption using utilization-aware compression. We evaluate our scheme's impact on link bandwidth and consequent system performance and energy, on a multi-module system where each module consists of streaming multiprocessors (SMs) and in-package memory in the form of Hybrid Memory Cubes (HMCs) as shown in Fig. 8. We use GPGPUSim v3.2.2 [16] for our experiments. GPGPUSim is coupled with GPUWattch [17] for modeling SM energy consumption and DRAMPower [18] to model the memory and I/O power. We configure GPGPUSim to model 3D stacked memory according to HMC specifications [19] since HMC allows integration of units into a network. The capacity of each stack is the same and total memory is set to be equal to the memory footprint of each application evaluated. The simulator parameters are summarized in Table I. The benchmarks evaluated are shown in Table II. The benchmarks are simulated to completion or till 1 billion instructions whichever occurred first.

We first compare the effectiveness of our prediction mechanism against PC-based prediction scheme used in [11]. We limit to PC-based schemes so that the patterns learnt from a warp can be used across all the warps. The responses are pruned to bring only the predicted bytes. No compression is applied. Fig. 9 shows the off-chip traffic observed using proposed granularity prediction and prior work. We can observe that our scheme performs better with average 22% (up to 56%) lower data movement. The prior work uses critical word index

TABLE II: Benchmarks.

Benchmark	Input
Backprop (BP) [21]	256k
Bfs [21]	1MW
Cfd [21]	0.2M
Heartwall (HW) [21]	test.avi, 5
Lulesh (LUL) [22]	Default
Mummer (MUM) [21]	NC_003997_q25bp.50k.fna
Gaussian (GAUS) [21]	matrix512
Sradv1 (SRAD) [21]	100, 0.5, 502, 458
Streamcluster (SC) [21]	Default

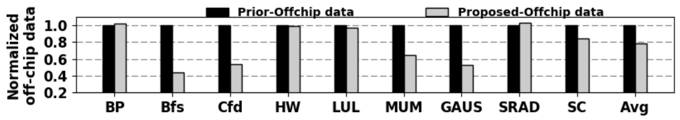


Fig. 9: Prediction mechanism performance.

on top of PC for indexing, resulting in conservative behavior with lower benefits. Our aggressive fine grained prediction results in higher data movement for *Backprop* and *Sradv1* but the increase is small (<4%).

We compare the reduction in actual off-chip data (not including padding bytes) moved under different configurations as shown in Fig. 10. The baseline (*Base*) has no compression and accesses complete cachelines. *Comp* uses traditional compression (C-PACK). *CompMask* uses granularity prediction and compresses the pruned data using traditional compression scheme and *CompMask-Lazy* uses granularity prediction and employs slack-aware compression on pruned response. *RQ* and *RSP* indicate data traffic type: memory request and response respectively. We can see that on average *Comp* reduces 28% of total offchip data traffic, *CompMask* gets higher (33%) reduction indicating spatial utilization can be exploited to obtain bandwidth saving. The effectiveness of lazy batch compression is evident from the average 42% traffic reduction obtained by *CompMask-Lazy*. Moreover, *CompMask-Lazy* reduces 12% higher off-chip traffic than *CompMask* indicating the additional compression opportunities captured by our lazy batch compression scheme. The request traffic for all configurations is in general higher than *Base* (except for *Comp*) due to the metadata (requested word mask (4 bytes) and granularity (1 byte)) overhead. This overhead is often compensated by the reduction in the response sizes resulting in net reduction in traffic. However, there are some cases when the net traffic increases e.g., *Heartwall and Backprop* for *CompMask*.

HMC uses packet switch interconnect and will have overhead of padding bytes if response sizes are not multiple of flit size. Schemes like [15] can be deployed to eliminate the same but has not been implemented in this work.

We now compare the system energy savings obtained with different configurations. I/O link energy saving methods have been proposed by *X. Jian et al.* [5], which save energy at the cost of bandwidth. Implementing these solutions is not the focus of this work and we assume that I/O component of link energy is reduced proportional to the reduction in link traffic. Fig.11 shows the normalized IPC and system energy across different configurations. Comparing the energy savings obtained we see that *Comp* reduces system energy by 12%

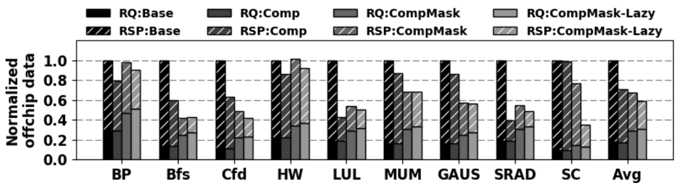


Fig. 10: Normalized off-chip traffic.

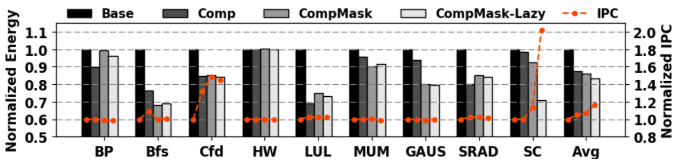


Fig. 11: Normalized IPC and energy saving.

while *CompMask* and *CompMask-Lazy* get 14% and 17% energy saving respectively. *CompMask-Lazy* reduces 4% (up to 24%) more system energy than *CompMask* scheme.

On comparing the IPC improvement across configurations (see Fig. 11), we see in general that the reduction in traffic is not reflected in IPC benefits, this is because, using a flitsize of 68 bytes diminishes the benefit obtained due to padding overhead and lower bandwidth pressure. The response size is 128 bytes (data) + 8 bytes (header) resulting in 2 flits per response. *Cfd and Streamcluster* still benefit from traffic reduction. On average *Comp* provides 5% (up to 32%) performance improvement, while *CompMask* and *CompMask-Lazy* provide 7% (up to 48%) and 16% (up to 2x) improvement respectively. *Streamcluster* benefits significantly due to its high data-content locality. We study the performance impact under smaller flit sizes. We reduce the flit size from 68B to 32B while keeping other network parameters same, this reduces the padding overhead and increases the bandwidth demand since every response now requires 5 flits. Fig.12 shows the IPC under different flit sizes normalized to 68B (*Base-68*). We can see that performance drops for many benchmarks due to increased bandwidth demand with 32B flits and in most cases our scheme is able to accommodate the increased bandwidth demand while achieving similar or higher performance. We find that under higher bandwidth demand *Comp* provides 20% average improvement while *CompMask* and *CompMask-Lazy* provide 23% and 36% improvement respectively.

VI. Conclusion

This paper presents CUALiT - Cacheline Utilization Aware Link Traffic Compression. The scheme exploits the variation in temporal and spatial utilization to achieve higher compression ratio. CUALiT uses a novel granularity predictor to predict the cacheline utilization at a very fine granularity. This information is then used to prune unutilized words, batch and lazily compress words with slack to harness data content locality across cachelines. Our evaluations show that CUALiT reduces the link traffic by 42% on average, subsequently reducing system energy by 17% and improving performance by 16%. Further, under high bandwidth demand, CUALiT provides 36% improvement demonstrating its effectiveness to reduce bandwidth demand.

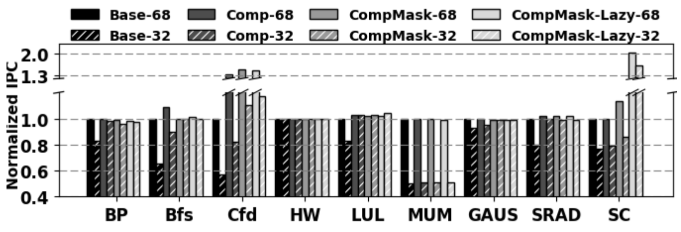


Fig. 12: IPC with 68B and 32B flitsizes.

VII. Acknowledgements

The authors would like to thank the reviewers for their comments. This work was partially supported by the National Science Foundation (NSF) under grant CCF-1725743.

References

[1] H. Jang *et al.*, "Bandwidth-efficient on-chip interconnect designs for GPGPUs," in *DAC*, 2015.

[2] U. Milic *et al.*, "Beyond the socket: Numa-aware gpus," in *MICRO*, 2017.

[3] T. Vijayaraghavan *et al.*, "Design and analysis of an apu for exascale computing," in *HPCA*, 2017.

[4] M. Poremba *et al.*, "There and back again: Optimizing the interconnect in networks of memory cubes," in *ISCA*, 2017.

[5] X. Jian *et al.*, "Understanding and optimizing power consumption in memory networks," in *HPCA*, 2017.

[6] G. Pekhimenko *et al.*, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *PACT*, 2012.

[7] J. Yang *et al.*, "Frequent value compression in data caches," in *MICRO*, 2000.

[8] V. Sathish *et al.*, "Lossless and lossy memory i/o link compression for improving performance of gpgpu workloads," in *PACT*, 2012.

[9] C. Huang and V. Nagarajan, "Increasing cache capacity via critical-words-only cache," in *ICCD*, 2014.

[10] M. Rhu *et al.*, "A locality-aware memory hierarchy for energy-efficient GPU architectures," in *MICRO*, 2013.

[11] S. Kumar *et al.*, "Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy," in *MICRO*, 2012.

[12] A. Arunkumar *et al.*, "LATTE-CC: Latency tolerance aware adaptive cache compression management for energy efficient GPUs," in *HPCA*, 2018.

[13] B. Panda and A. Seznec, "Dictionary sharing: An efficient cache compression scheme for compressed caches," in *MICRO*, 2016.

[14] X. Chen *et al.*, "C-Pack: A high-performance microprocessor cache compression algorithm," *IEEE Trans. on VLSI Systems*, 2010.

[15] J. Kim *et al.*, "Bit-plane compression: Transforming data for better compression in many-core architectures," in *ISCA*, 2016.

[16] A. Bakhoda *et al.*, "Analyzing cuda workloads using a detailed gpu simulator," in *ISPASS*, 2009.

[17] J. Leng *et al.*, "GPUWattch: enabling energy optimizations in GPGPUs," in *ISCA*, 2013.

[18] K. Chandrasekar *et al.*, "System and circuit level power modeling of energy-efficient 3D-stacked wide I/O DRAMs," in *DATE*, 2013.

[19] HMC Consortium, "Hybrid memory cube spec. 2.1," tech. rep., 2015.

[20] K. Hsieh *et al.*, "Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems," in *ISCA*, 2016.

[21] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.

[22] I. Karlin *et al.*, "Lulesh programming model and performance ports overview," tech. rep., Lawrence Livermore National Lab, 2012.