A Survey Of Architectural Approaches for Data Compression in Cache and Main Memory Systems

Sparsh Mittal, *Member, IEEE,* and Jeffrey S. Vetter, *Senior Member, IEEE*

Abstract—As the number of cores on a chip increase and key applications become even more data-intensive, memory systems in modern processors have to deal with increasingly large amount of data. In face of such challenges, data compression presents as a promising approach to increase effective memory system capacity and also provide performance and energy advantages. This paper presents a survey of techniques for using compression in cache and main memory systems. It also classifies the techniques based on key parameters to highlight their similarities and differences. It discusses compression in CPUs and GPUs, conventional and non-volatile memory (NVM) systems, and 2D and 3D memory systems. We hope that this survey will help the researchers in gaining insight into the potential role of compression approach in memory components of future extreme-scale systems.

Index Terms—Review, classification, cache, main memory, compaction, compression, data redundancy, non-volatile memory, 3D memory, extreme-scale computing systems.

1 Introduction

Recent years have witnessed an exponential growth in the amount of data produced by scientific applications and social media. Modern data-centers contain up to tens of thousands of servers which provide 24X7 service to thousands of users [1]. It is expected that future exascale systems, capable of executing 10^{18} operations/second, will deal with hundreds of exabytes of data and will require at least 100 petabytes of main memory. Scaling the power consumption trends from today's DDR3 DRAM, a main memory of this capacity itself would consume 52MW power [2], which is much more than the 20MW power budget allocated to the entire system. It is clear that, over-provisioning of memory system resources alone cannot solve the rising demands of data processing; in fact, efficient approaches for storing and processing data will be an integral part of future memory systems [3].

Data compression is a promising approach for increasing effective memory system capacity without incurring the area/energy overheads of a large-sized memory. Compression can reduce miss rates and improve bandwidth usage, which translates into performance and energy efficiency improvements [4]. Thus, due to its advantages, compression has been used for both cache and main memory. However, compression and decompression steps themselves incur latency, energy and area overheads, which may nullify the advantage of compression or introduce significant additional complexity. Thus, effective use of compression approach requires a careful choice of compression algorithm, selection of parameters, and insight into compressibility of data etc.

Several recent works have proposed techniques to address these challenges. In this paper, we present a survey of architectural approaches which leverage data compression in cache and main memory systems. We classify these techniques based on several key parameters. Since different techniques have been evaluated on different platforms/workloads, we do not discuss their quantitative results, instead, we focus on the qualitative results to get insight. To provide application perspective, we also discuss a few compression techniques used in real processors (Section 5.10). The aim of this paper is to help the designers and researchers in understanding state-of-the-art in compression and motivate them to futher improve the effectiveness and applicability of compression in modern computing systems.

The remainder of this paper is organized as follows. Section 2 discusses the advantages and scope of compression, along with tradeoffs involved in its use. Section 3 presents a classification of compression techniques based on several parameters. Section 4 discusses compression algorithms and Section 5 reviews several architectural and system-level techniques which use compression. Finally, Section 6 concludes this paper.

• *The authors are with the Future Technologies Group at Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA 37830. Jeffrey S. Vetter is also with Georgia Institute of Technology, GA, USA. E-mail: {mittals,vetter}@ornl.gov*

2 Background

2.1 Benefits of compression

With increasing number of cores on a chip, the demand of cache and main memory capacity is on rise [5]. However, due to energy and bandwidth constraints, using large-size memory systems becomes infeasible and this has led to decreasing memory-to-core ratio in recent processors. Compression can help in storing the data in smaller amount of physical memory, thus, giving the impression of a large size memory to the application or end-user. Cache compression can reduce costly off-chip accesses and memory compression can reduce page faults which trigger disk accesses [6, 7]. Compression also helps in reducing the memory bandwidth requirement, since multiple consecutive compressed blocks can be fetched at the cost of accessing a single uncompressed block.

Further, compression reduces the footprint and memory requirement of an application and hence, it may allow transitioning the unused portions of memory into low power states for saving energy [8–11]. The reduced energy consumption and temperature may alleviate the need of thermal throttling, which provides headroom for achieving higher performance within same power and temperature constraints [12]. Clearly, the performance improvement of a compression technique may come from any one or combination of factors, viz. miss-rate reduction, bandwidth reduction, energy saving which permits more computations within same power budget etc.

Compression approach is especially promising for extreme-scale computing systems. For ensuring resilience, these systems use checkpoint/restart and due to their scale of operation, data movement within and across node incurs prohibitive latency and energy overheads. Also, these systems generally feature CPU-GPU heterogeneous architecture where data transfer happens over a connection bus; this, however, presents the most crucial bottleneck in achieving high application performance. Compression can mitigate these overheads and also improve cost efficiency by reducing the footprint of data. Further, achieving exascale performance with a power budget of 20MW requires energy efficiency of 50 Gflops/W [3], while the most energy efficient computer in November 2014 Green500 list has energy efficiency of 5.27 Gflops/W [13]. Clearly, compression can be an integral part of the suite of techniques used to bridge this magnitude order gap in energy efficiency [4].

2.2 Opportunities for compression

The data accessed by real-world applications show significant amount of redundancy which provides opportunity for compression. Such redundancy may arise due to use of constants, nature of program inputs and operations such as assignment and copying. For example, several applications use a common value for initializing a large array and in an image processing application, multiple adjacent pixels may have the same color. Similarly, on using `memcpy()` two copies of data may be stored in the cache.

Due to these factors, some benchmarks can contain up to 98% of duplicated blocks [14]. Further, it has been shown that if each unique memory value is stored exactly once in the cache, compression ratios beyond 16X can be achieved [15]. However, due to practical considerations (discussed below), most practical techniques achieve compression ratios of less than 2×, although a few techniques achieve compression ratios of 3-4× [15, 16].

In several cases, programmers provision large size data types to handle worst-case scenario, while most values can fit in a smaller data type, for example, a four-byte integer may store only a one-byte value. Such values, referred to as narrow-width values, can be easily compressed. Similarly, 'special patterns', such as data values where all the bits are either zero or one, can be coded using special flags or smaller number of bits. Further, in many cases, the differences between values stored within the cache line may be small and hence, they can be represented in a compressed form using a base value and an array of differences.

2.3 Factors and tradeoffs in compression

The efficacy of compression approach depends on several factors. In what follows, we discuss some of these factors.

Compressibility of data: Different applications and data have different compressibility, which can greatly affect the improvements obtained from the compression approach. In case of incompressible data or inefficient compression algorithm, the size of a compressed block may even be larger than that of uncompressed block due to use of metadata etc. For such cases, storing data in uncompressed form is beneficial. For example, compression approach exploits data redundancy, and hence, it may not benefit systems which use encrypted data since data encryption techniques introduce randomness and reduce data redundancy [17]. Same is also true for using compression with redundant bit write removal (RBR) in NVM systems (refer Section 5.7).

Choice of data granularity: Larger block sizes are expected to compress well due to higher redundancy, however, at large sizes, access to even a single sub-block will require decompressing the whole block. On the other hand, for some compression algorithms such as zero-content detection, use of smaller blocks may lead to larger compression ratios since finding certain patterns may be easier. However, use of smaller block sizes may lead to larger overhead of metadata.

Choice of compression algorithm: In general, the compression algorithms which provide high compressibility (e.g. Lempel-Ziv and Huffman encoding) also have high latency/energy/area overhead [16] and due to variable length compression, they may not allow parallel decompression. Conversely, the algorithms which exploit certain data patterns may not be universally applicable.

Latency and energy overhead: Compression and decompression incur latency/energy overhead and require additional hardware or software routines. The data when written in compressed form need to be decompressed for reading. Thus, while compression can take place in the 'background', decompression lies on the critical memory-access path. This extra latency can negate the capacity benefits of compression [18–20]. To partially mitigate the overhead of decompression, recently decompressed data can be cached [8, 19, 21, 22] which avoids the need of decompressing the data again.

Implementation complexity: In general, compression significantly increases the complexity of memory management. Since different blocks show different compressibility, the compressed blocks show variable size which can break linear mapping between tag and data and cause large overhead of metadata [23]. Due to this, finding a cache block within a set may require calculating the offset using the sizes of all the blocks stored before it, which may increase the block lookup time. Also, variation in compressed block size leads to data fragmentation [24, 25], since the length of evicted and inserted data in compressed cache may be different, which may necessitate compaction to create contiguous free space.

Similarly, on using main memory compression, fixed-length virtual pages need to be mapped to variable-length physical pages. Since physical addresses are unique, but virtual addresses are not, modern processors compute cache tags from physical addresses to avoid aliasing. Hence, use of memory compression requires modification of cache tagging logic to avoid computing the memory address on critical path of cache access. In the similar vein, use of bandwidth (link) compression requires support for variable-size compressed data in both off-chip memory controller and on-chip memory interface. This also breaks the assumption of a generic memory interface that allows plug and play capability for off-chip components from different vendors.

Also, when the working set size (WSS) of the application is very large, a small capacity increase provided by compression may be insufficient and hence, compression may need to be combined with other techniques, thus leading to increased complexity. Conversely, if the WSS of an application is small, the additional capacity provided by compression may not reduce its miss-rate.

Properties of processor architecture: While the CPU architecture is oriented towards optimizing latency, the GPU architecture is oriented towards optimizing throughput and thus, they present different tradeoffs in design of compression approaches. For example, since GPUs demand high bandwidth, bandwidth compression approaches are even more important for them than they are for CPUs. Hence, conventional compression techniques proposed in context of CPUs may need to be re-evaluated and optimized for use in GPUs. Also, in both fused and discrete CPU-GPU heterogeneous systems, compression in one PU (processing unit) needs to be performed while taking into account the characteristics and constraints of the other PU.

Properties of cache level: Since first-level caches are typically optimized for latency and not for capacity [5, 26], use of compression in these caches requires a careful control of aggressiveness of compression [27]. In general, compression is used in lower-level caches and main memory, since a small increase in their latency can be more easily tolerated by modern processors [5, 11].

Properties of underlying memory technology: Recently, non-volatile memories (NVMs), such as phase change memory (PCM), STT-RAM (spin transfer torque RAM) and ReRAM (resistive RAM) have been used for designing cache and main memory due to their several attractive properties [3, 26, 28, 29]. A common characteristic of these memories is that write to them incur significantly large overhead than reads and their write endurance is orders of magnitude smaller than that of SRAM and DRAM [26, 30]. In general, by reducing the amount of data written to memory system, compression may improve the performance and lifetime of NVMs. However, (de)compression can generate additional write operations [31], which may harm NVM lifetime. For these reasons, careful management is required to realize the full potential of compression in NVMs.

Merits relative to other techniques: In addition to compression, researchers have also explored other techniques to exploit data redundancy and save cache/main-memory energy. These techniques include data encoding [26, 32], cache reconfiguration [33] etc. For example, Bojnordi et al. [32] present a technique to save energy of data exchange in LLC interconnect by reducing the activity factor (i.e. switching probability) of the wires. Their technique represents information by the delay between two successive pulses on a communication wire. This limits switching events to one per transmission, independent of the data patterns. Similarly, cache reconfiguration techniques work by dynamically turning-off a portion of the cache based on the application working set size to save leakage energy. Thus, the decision of using compression in a product system depends on its relative advantage over alternative techniques.

From above discussion, it is clear that the compres-

sion approach, although promising, is not a panacea and hence, many techniques use compression in a selective (or adaptive) manner [8, 18, 20, 21, 27, 28, 34–38].

3 Classification and Overview

Table 1 presents a classification of different techniques. This table classifies the techniques based on the processor component in which compression is used, the algorithm used to compress data and the objective (or figure of merit). Note that while an improvement in performance may also lead to improved energy efficiency, in this table we classify a technique based on the figure of merit for which it was actually evaluated. Finally, the table also summarizes use of compression in different contexts or in conjunction with other approaches.

From the table, it is clear that this paper covers a broad range of techniques. The techniques discussed in the paper use compression algorithms ranging from general purpose such as Lempel-Ziv (LZ) algorithm [39] to those based on specific patterns, e.g. narrow values [40] or zero values [41] etc. Some techniques perform hardware compression (e.g. [35, 40]), while others perform compression using software (e.g. [36, 42]) or using compiler (e.g. [43, 44]). Finally, these techniques have been used or evaluated in systems ranging from embedded systems [7, 36, 45–47] to high-end servers e.g. [22, 48].

TABLE 1
A classification of research works

Classification	References
Processor Component	
Cache compression	[4, 8, 10, 14, 16, 18, 21, 24, 27, 31, 34, 35, 41, 43, 49–61, 61–73]
Memory compression	[6, 7, 9, 12, 17, 20, 22, 23, 28, 29, 36–38, 42, 44–46, 68, 69, 74–92]
Both cache and memory	[19, 21, 48, 81, 85, 86]
Compression algorithms used/evaluated	
Frequent pattern compression (FPC)	[8, 10, 12, 16, 17, 23, 28, 29, 35, 40, 48–50, 52, 53, 56, 63, 64, 66, 78, 82, 83, 88, 93]
Zero-content compression	[14, 35, 41, 50, 52, 57, 62, 78, 81]
Frequent value compression (FVC)	[35, 52, 59, 60, 67, 70]
Base delta immediate (BDI) compression	[16, 23, 35, 48, 51, 63, 83, 93, 94]
LZ and variants	[6, 7, 9, 17, 19, 20, 22, 34, 36, 38, 44, 46, 49, 65, 67, 74, 75, 77, 78, 90]
X-match and X-RL	[21, 49, 65, 78, 87]
Huffman coding	[15, 16, 85]
C-pack	[49, 50, 62, 89, 93]
SAMC (semi adaptive Markov compression)	[47, 68]
Others	[67, 86]
Objective	
Performance improvement	[6, 12, 14, 16, 18, 19, 23, 28, 35, 38, 43, 46, 48, 50, 51, 53, 56, 57, 62, 66, 69, 73–77, 82, 83, 85, 89–93]
Energy saving	[8–10, 16, 23, 24, 28, 31, 41, 43, 46, 48, 50, 51, 60, 62, 63, 66, 72, 82–85, 91, 93, 94]
Capacity increase and miss rate reduction	[7, 16, 18, 22, 23, 27, 35, 43, 47, 48, 50, 56, 57, 60, 62, 68, 69, 76–78, 81, 90, 91]
Bandwidth reduction	[16, 23, 35, 48, 56, 57, 60, 63, 67, 89, 93]
NVM write reduction	[17, 28, 29, 31, 88]
Thermal management	[12]
Interaction with other approaches and use in other contexts	
Prefetching	[16, 23, 48, 56, 58, 61, 71]
Resource-scaling and energy saving	[8–10, 59, 84, 85, 94]
Error-correction	[51, 83]
Link compression	[4, 19, 56, 89, 93]
Compression in NVM	[17, 28, 29, 31, 82, 88]
Compression in 3D cache or main memory	[8, 12]
Compression in GPUs	[89, 93, 94]
Use of compiler	[9, 21, 43, 44, 72, 73, 93]

4 Compression Algorithms

In this section, we discuss compression algorithms which are used in different research works. Note that the basic ideas of these algorithms are not mutually exclusive and hence, some algorithms can be considered as special cases of other algorithms, e.g. algorithms which recognize zero blocks [57] can be seen as special cases of frequent pattern compression [40].

It is noteworthy that different approaches exploit redundancy (or correlations or specific data patterns) at different levels, for example, deduplication techniques (e.g. [14]) remove redundant values across *different blocks of the whole cache*, while some techniques (e.g. [29, 35, 82]) exploit redundancy across *different words of a cache block* and some other techniques (e.g. [40]) exploit redundancy across *different bytes of a cache word*. On the other hand, redundant bit-write removal (RBR) techniques which compute bit-level difference between the incoming value with originally stored value and write only changed bits (e.g. [17]), exploit redundancy across temporal dimension. Some techniques (e.g. [28]) first compute the bit-level difference and then apply compression to achieve higher compression ratio.

Huffman coding works by analyzing the data to determine the probability of different elements. Afterwards, the most probable elements are coded with fewer number of bits than those which are least probable. Thus, Huffman coding uses a variable-length coding scheme. Lempel Ziv (LZ) compression algorithm works by replacing repeated occurrences of data elements with references to a single copy of that element existing earlier in the uncompressed data. Several variants of it have been proposed in the literature, which are used by various techniques.

X-match hardware compression algorithm [87] uses a dictionary and replaces each input data element with a shorter code whenever a complete or partial match with a dictionary entry is found. The X-RL algorithm extends the X-match algorithm by noting that consecutive zeros are common in memory-data. Hence, in the X-RL algorithm, before applying the X-match algorithm, the length of consecutive zeros

is also encoded, which improves the efficiency of compression, especially for memory data containing frequent consecutive zeros.

Yang et al. [60] present frequent value compression (FVC), which works on the observation that few distinct values occupy a large number of memory accesses during program execution and hence, data compression can be achieved by encoding these frequent values. Each cache line can hold either an uncompressed block or two compressed blocks as long as the size of each is no more than half of the size of the uncompressed block. This algorithm compresses both normal and narrow width data, but the limitation of this is that with increasing cache block size, the probability of finding frequent values decreases. Also, finding the frequent values incurs profiling overhead.

Alameldeen et al. [40] present a significance-based compression algorithm, called frequent pattern compression (FPC), which works on the observation that most data types can be stored in a smaller number of bits than the maximum allowed. For example, when an 8-bit number is stored in a 64-bit integer, all the information of the word is stored in the least significant bits. This algorithm scans each word to detect whether it falls into one of the patterns that can be stored in a smaller number of bits, and stores it in a compressed form with a suitable prefix.

Dusser et al. [57] present a design which augments a conventional cache with an additional cache, called zero-content (ZC) cache, for memorizing null (zero) blocks. Since null blocks exhibit high spatial locality, several null blocks are associated with a single tag in the ZC cache, which achieves high compression ratio. Their design requires very simple (de)compression circuits and avoids writes where null blocks are overwritten with null data. The limitation of their approach is that it cannot compress data with other patterns, e.g. narrow-width data. Dusser et al. [81] apply null data compression in main memory and also show that by combining their design with ZC-cache design provides an opportunity to manage null blocks throughout the whole memory hierarchy.

Kim et al. [24] exploit narrow sign-extended values by compressing the upper portion of a word to a single bit if all the bits in it are zero or one. They store the lower portions of the words and the compressed sign bits in a bank. These are accessed first, assuming that the data are compressed. When the data are not compressed, the remaining cache line is accessed. They also propose using a second tag with each line to allow storage of compressed cache lines in both half ways. This helps in increasing the effective cache size, at the cost of using extra tags, sign extension bits and the incompressible upper half-word storage for both ways of the cache line.

Chen et al. [49] present C-PACK, which is a pattern-based partial dictionary match compression algorithm. C-PACK compresses data by both statically and dynamically detecting frequently appearing data words. Dynamic updating of dictionary allows adaptation to those frequently appearing words which are not captured by statically encoded patterns. Both partial and full word-matching are supported by the dictionary. For organizing compressed blocks in the cache, they use 'pair-matching', where a compressed line can be placed in the same line with a partner only when their total size is less than that of an uncompressed line. Pair-matching simplifies management of the locations of the compressed lines.

Pekhimenko et al. [35] propose base-delta-immediate (BDI) compression, which works on the observation that for many cache lines, the differences between values stored within the cache line are small. Using this, a cache line can be represented using a base value and an array of differences, such that their combined size is much smaller than that of the original cache line. However, several cache lines cannot be easily compressed using this simple approach. To further extend the applicability of their compression algorithm, they propose using two bases, such that one base is always set to zero. This helps to compress both wide and narrow values separately.

5 Architectural Techniques Using Compression

We now review several techniques which use compression. For convenience, we roughly classify them under several categories.

5.1 Techniques for cache

Alameldeen et al. [18] propose an adaptive cache compression technique which uses FPC algorithm [40]. In each of its sets, the 8-way LLC can store up to eight compressed lines but has space for only four uncompressed lines. On each LLC access, the LRU stack depth and compressed block size are used to decide whether compression eliminated or could have eliminated a miss or whether the access would hit or miss regardless of compression. Based on this analysis, their technique updates a global counter which weighs the cost and benefit of compression and makes decision about allocating a block in compressed or uncompressed form. They show that their technique provides better performance than a technique which always performs compression.

Nitta et al. [67] use cache compression to increase effective data bandwidth. They classify the cache lines into four types, viz. integer, floating point, pointer and other, based on whether 50% of more of the values in the cache line belong to that type. Since each of these types have different compressibility, they apply different compression schemes for each classification. They have shown that their technique reduces the

bandwidth requirement effectively with lower complexity than other techniques, such as Lempel-Ziv-Welch algorithm.

Zhang et al. [70] augment the L1 cache with a 'frequent value cache' (FVC) which holds only frequently occurring values. FVC uses compact encoded representation of frequent values to achieve data compression. The fact that the number of highly frequent values is small, enables using only a small size FVC. FVC helps in keeping more data on-chip which reduces off-chip traffic and improves energy efficiency. However, for large sized caches, the size of FVC also increases and its access time and energy consumption no longer remain negligible.

Pujara et al. [27] propose a selective compression approach for L1 cache which aims to minimize latency increase due to compression. Their technique compresses a cache block only if all the words in the block are of narrow width. Since in this technique, a block will not be compressed even when a single word is of normal-width, they also discuss an extension of this technique. In this technique, additional half-words storage is provided for each physical cache block to store the higher order half-words of the few normal-width words. This allows compressing even those blocks which have a small number of normal width words. They also propose a method to reduce the requirement of extra tag bits that accompanies a compression technique. They note that the higher order bits of the address tags in a set are expected to be the same, hence, instead of providing all the bits for additional tags, only a small number of extra bits need to be provided for each extra tag.

In multicore processors, cache is shared by multiple applications which benefit differently from cache compression and hence, a thread-oblivious policy can lead to non-uniform speedups on multicore processors. Towards this, Xie et al. [53] propose a technique which accounts for thread-criticality information to compress/decompress data at the runtime. For each thread, their scheme estimates the performance impact of different compression decisions and uses this to make decision about compressing data for the thread.

Pekhimenko et al. [63] propose management policies for caches that use data compression. They note that it is beneficial to keep multiple compressed blocks in cache and evict a larger, potentially uncompressed block (of size same as total size of compressed blocks), if it helps in increasing cumulative hits in the set. Thus, their eviction policy considers both locality and size of the block to make eviction decisions. They also note that, elements belonging to the same data structure within an application sometimes lead to same sized compressed blocks and hence, the compressed size of a cache block can sometimes be used as an indicator of its reuse characteristics. Based on this, they propose a cache insertion policy that dynamically prioritizes cache blocks using their compressed size as an indicator of data reuse of the block. By combining these two policies, their approach achieves significant performance and energy efficiency improvements and memory bandwidth reductions.

Sardashti et al. [50] propose a cache management scheme for using compression to improve both the performance and energy-efficiency. Their technique, termed DCC (decoupled compressed cache) uses superblocks, which refers to four aligned contiguous cache blocks that share a single address tag. Each 64-byte block in a super-block is compressed and then compacted into multiple 16-byte sub-blocks. To reduce fragmentation within a super-block, DCC decouples the address tags and thus, any subblock in a set can map to any tag in that set. Decoupling also allows the subblocks of a block to be non-contiguous which eliminates the overhead of recompaction whenever the size of a compressed block changes. The decoupling, however, also introduces extra metadata and latency since locating a block requires additional pointers.

Sardashti et al. [62] also propose skewed compressed cache (SCC) which uses superblocks to reduce tag overhead and fragmentation but overcomes the limitation of DCC of extra metadata and latency by retaining a direct tag-data mapping. Their technique uses sparse super-block tags which can track anywhere between one block to all blocks in a superblock, depending on their compressibility. The limitation of this direct tag-data mapping is that more than one tags are required to map blocks from the same superblock. To minimize conflicts between blocks, SCC uses two forms of skewing. First, it maps blocks to different cache ways based on their compressibility. For each cache way, different hash functions are used which are functions of the block address. Second, SCC also skews compressed blocks across sets within a cache way to decrease conflicts. Such skewed-associative mapping allows fast lookup with low metadata overhead even for variable-sized compressed blocks.

Arelakis et al. [16] present a cache design which uses Huffman-based statistical compression to achieve high compression ratios. Compression algorithms such as Huffman coding provide higher compression ratios, but also incur high (de)compression overheads and require a sampling phase for collecting statistics for code generation, and hence, such algorithms are not widely used. They note that there is little variation in value-locality over time and across applications and hence, aggressive compression algorithms can be used since new encodings are needed rarely and thus, the task of statistics acquisition can be offloaded to software routines. Using this, the compression latency can be kept off the critical memory access path and the effect of decompression latency on performance becomes negligible. They show that their technique provides performance advantage of a 4X size cache

with much smaller power overheads.

Das et al. [66] evaluate both cache compression (CC) and network compression (NC). In CC, the data are compressed after the L1 cache, prior to sending it to a shared L2 cache bank. This provides both storage and communication compression. In NC, compression is performed at the network interface controller (NIC), before sending the message to the network. In NC, only network communication traffic is compressed. Since NC scheme does not require any changes to the L2 cache banks, it can be easily integrated into a system-on-chip. The limitation of NC is that it requires a (de)compressor unit in the NIC of every node in the network and (de)compression is performed for both read and write requests. By comparison, CC requires (de)compressor unit only in the CPU nodes and involves only one operation for a given request, viz. compression for writes and decompression for reads. They show that both CC and NC can provide significant reduction in network latency and energy consumption.

Data deduplication [14] is a technique which eliminates duplicated copies of repeating data and stores the redundant data as references to a single copy in a deduplicated data storage. Thus, deduplication is a specific compression approach. Tian et al. [14] propose a data deduplication technique for LLCs which stores a single copy of duplicated data in a manner that can be accessed through multiple physical addresses. Decoupling of tags allows multiple tags to be associated with a single data-block. Deduplication is performed at block level and duplication of values is detected using augmented hashing. They show that their technique achieves larger capacity improvement than compression techniques based on detecting zero-content and frequent patterns [18, 57].

5.2 Techniques for main memory

Wilson et al. [38] use compression in virtual memory management scheme to compress portions of main memory (called 'compression cache') to reduce page faults. Based on the cost-benefit analysis of compression, their technique dynamically adjusts the size of compressed and uncompressed regions. Their technique tracks the program behavior by counting the number of touches to pages in different regions of the LRU ordering, and uses this to estimate hits or misses for different partitions of compressed/uncompressed regions. This partitioning decides the fraction of pages that will be compressed by their technique.

The IBM MXT (memory expansion technology) [22] technique uses compression to reduce the number of main memory accesses. It adds a large (32MB) L3 cache which stores uncompressed data and is managed at the granularity at which blocks are compressed (1KB). The data are compressed before being written to main memory, using a variant of LZ compression algorithm. On a memory access, the block is decompressed using a parallel implementation on decompressor hardware on a special-purpose ASIC (application-specific integrated circuit) which takes tens of cycles. Also, this technique uses a memory-resident table, which provides address indirection for locating a compressed block and is consulted on each memory access. This, however, significantly increases the memory access latency. The L3 cache partially hides such large latencies, however, if the working set size increases L3 capacity, the performance loss may become large.

Ekman et al. [78] propose a main memory compression approach which addresses limitations of MXT technique, viz. indirect access, and performance loss. They use low-latency compression algorithm, viz. FPC [40], and show that even a simple zero-content compression approach can provide significant compression ratios. To efficiently locate a block, they keep address translations in a TLB-like structure which allows address translations to be carried out in parallel with LLC access. To reduce fragmentation caused by compression, relocation of parts of the compressed region is performed, and the frequency of this is kept low to minimize its impact on performance.

Nakar et al. [37] present a selective compression technique for main memory. Their technique identifies the current working set and leaves it uncompressed for allowing faster access. Also, the pages which are not in the working set are compressed during idle time of CPU. Their technique also identifies program phases by detecting changes in the program's working set. Based on this, compression is performed only during those phases that exhibit good spatial locality.

Chen et al. [84] use compression to save energy in embedded Java environments. They store the code of the embedded JVM (Java virtual machine) system and associated class libraries in compressed form in the memory. This reduces the memory requirement of the system. Further, using a power-gating scheme, the unused portions are turned-off to save leakage energy. When a compressed code is required, a mapping structure stored in SPM (scratch pad memory) locates the required block and then, the decompressed block is brought in the SPM. Due to its smaller size, SPM has smaller access energy and thus, their technique saves both leakage and dynamic energy.

Yang et al. [36] present a software-based memory compression technique for embedded systems. Their technique divides the RAM into two portions, one containing compressed data pages and the other containing uncompressed data pages and code pages. Using OS virtual memory swapping mechanism, their technique decides which data pages should be compressed and when the compression should be performed. When required, their technique compresses and moves some of the pages from the uncompressed portion to the compressed portion to increase the memory capacity. When a compressed page is later

required, it is located, decompressed and copied to the uncompressed portion. Their technique performs compression only for those applications which may gain performance or energy benefits from it. They have shown that increased effective memory capacity provided by their technique enables several applications to execute with reasonable performance/energy penalties, which without their technique, may suffer from performance degradation or instability.

Benini et al. [46] present a memory compression approach for reducing the memory traffic. In their approach, cache lines are compressed before being written back to main memory and decompressed when cache refills take place. They study both static and dynamic data-profiling based compression schemes, where the static scheme is more suitable for embedded systems while the dynamic scheme is suitable for systems where several programs need to be executed and offline profiling provides limited utility. They study different compression algorithms which represent different trade-offs between compression ratio and performance overheads.

5.3 Techniques for both cache and main memory

Hallnor et al. [19] propose a compression approach for both LLC and main memory. They use an indirect index cache where decoupling of tags allows fully-associative placement of data. A tag is not associated with a specific data block, instead, each tag holds a pointer to a data array which contains the blocks. When the cache block is compressible, some of the subblock pointers are not used and the unused blocks can be associated with other tags. The L3 cache (LLC) stores compressed data and L2 cache acts as a buffer of decompressed blocks. Further, by using similar compression in both LLC and main memory, the data can be transferred between them in compressed form, which increases the effective memory bandwidth. The limitation of their technique is higher complexity of cache replacement policy. Also, decoupling of tags necessarily serializes tag and data array accesses.

Pekhimenko et al. [48] note that variable-size compression of cache lines within a memory page complicates address translation and incurs overhead in address computation. To remedy this, they propose compressing all the cache lines within a given page to the same size. This simplifies the computation of the physical address of the cache line to a mere shift operation. Using this idea, their technique determines a target compressed cache line size for each page. The cache lines that cannot be compressed to this target size are stored separately in the same compressed page, along with their metadata. They show that multiple cache compression algorithms [35, 40] can be adapted for use in their approach. They show the use of their approach for compressing data in main memory alone and in both cache and main memory.

5.4 Compiler based techniques

Embedded systems typically do not have disks and virtual memory and hence, out-of-memory errors in such systems lead to disastrous consequences. Hence, designers need to make a compile-time estimation of maximum memory requirement of running tasks which is challenging. To address this, Biswas et al. [7] propose using memory compression along with other approaches. First, the application code is modified in the compiler to insert software checks for out-of-memory conditions. Second, whenever stack or heap segments run out of memory, they are allowed to grow into non-contiguous free space in the system to reduce the application's memory footprint. Third, when all the free space has been used for managing overflow, live data are compressed to free more space for accommodating stack and heap overflows. The data are decompressed later when a memory access request arrives. Only those global data are chosen for compression that are not likely to be used in near future. Their technique uses compression only when system runs out of memory, where a relatively large overhead of (de)compression is accepted, since it avoids much worse consequences.

Zhang et al. [43] present a compression approach which works by applying transformations to dynamic data structures used in programs. They propose transformations for data structures that are partially compressible, that is, they compress portions of data structures to which transformations apply and provide a mechanism to handle the data that are not compressible. Their technique initially allocates data storage for a compressed data structure assuming that it is fully compressible. At runtime, when incompressible data are encountered, additional storage is allocated to handle such data. To allow efficient access to compressed data, they also propose additional instructions in the ISA (instruction set architecture). They have shown that use of their technique helps in achieving significant reduction in heap allocated storage, execution time and energy consumption.

Ozturk et al. [9] integrate data compression with other techniques to save energy in DRAM main memory by effectively transitioning DRAM banks into low power modes. They model DRAM energy minimization problem as an integer linear programming (ILP) problem and solve it using an ILP solver. Based on the application-data access patterns extracted by the compiler, the data replication scheme aims to increase the idle time of certain banks by duplicating their selected read-only data blocks on other active banks. Data compression helps in reducing the footprint of an application which reduces the number of memory banks occupied by data. Also, data migration aims to cluster data with similar access patterns in the same set of banks. The optimal data migration, compression, and replication strategies obtained by the ILP

solver are finally fed to the compiler, which modifies the application code to insert explicit migration, (de)compression, and replication calls (instructions).

Ozturk et al. [44] present an application-aware compression strategy which utilizes the control flow graph (CFG) representation of the embedded program. In their work, all the basic blocks of the application are assumed to be compressed in the beginning. Their technique decompresses only those blocks that are predicted to be required in the near future. Further, when the current access to a basic block is over, their technique decides the point at which the block can be compressed. Thus, compression helps in increasing the effective memory capacity. They also note that performing compression at basic block granularity, instead of function granularity helps in achieving higher memory space savings.

Lee et al. [21] present a selective compression technique where only the data blocks which compress to less than 50% of size are compressed. The selective compression is performed by the compiler for an executable file in post-compilation process and an information file is generated which shows whether each block is compressed or not. Their technique allocates fixed space to all compressed blocks, which leads to internal fragmentation, but provides the advantage of easier and efficient management. To hide the decompression latency, their technique allows the processor to access a decompressed critical word without waiting for the complete decompression of a whole block. Also, by overlapping the data transfer time with the decompression time, the overhead of decompression is hidden. They also use a decompression buffer which stores the recently decompressed blocks and thus, avoids the overhead of again decompressing a block.

Lattner et al. [73] note that compared to 32-bit pointers, 64-bit pointers can decrease performance by reducing cache/TLB capacity and memory bandwidth. However, even on a 64-bit machine, only a few *individual* data structures use more than 4GB memory since the primary use of pointers in most programs is to traverse linked data structures. Based on this, they present a compiler approach which converts pointers for selected data structures in a program to use smaller representation (eg. from 64-bit to 32-bits). Their technique modifies the program to allocate and free objects from memory pool in the system heap. Further, heap objects from different data structures are separated into different pools. At compile time, every static pointer variable pointing to the heap is mapped to a unique pool descriptor. Based on this, a 64-bit pointer is compressed by substituting it with a smaller integer index from the start of the corresponding pool. Their technique allows using compression for selected pools and thus, other pools can grow to full 2^{64} bytes. They show that their technique improves performance by reducing the memory consumption significantly.

5.5 Interaction with other techniques

Phalke et al. [71] note that high compressibility of application data implies high predictability of program characteristics. Thus, compression can be used as a tool for predicting program behavior for performing memory management tasks. Based on this, they study two program properties, viz. inter reference gap (IRG) and cache misses. IRG is defined as the time interval between successive references to the same address and based on its predictability, better replacement decisions are made to reduce the number of cache misses. Also, predictability of the sequence of cache misses is utilized for improving cache prefetching for the purpose of reducing cache misses.

Zhang et al. [61] propose a technique which uses cache compression to reduce memory bandwidth consumption and further utilizes the freed bandwidth to prefetch additional compressible values. With each cache line in memory, their technique associates another line which acts as the prefetch candidate. On a cache line prefetch, the compressibility of values in the cache line and the associated prefetch candidate line is examined. If the i-th word of the line and the i-th word from its prefetched candidate line are both compressible, both words are compressed and transferred which consumes the bandwidth of only one word. This is repeated for each pair of corresponding words in the two lines. Since the prefetched values are stored in the space freed in the cache by storing values in compressed form, their technique does not require prefetch buffers or cause cache pollution.

Chen et al. [51] propose a design that utilizes the unused fragments in the compressed cache design to store ECC and thus avoid the overhead of a dedicated ECC storage. This helps in improving cache utilization and energy efficiency. When the unused fragment is not large enough, a lightweight EDC (error detecting code) is embedded into the block and the ECC is stored in a separate reserved block of the same cache set. This ECC is accessed only if an error is detected by EDC checking, which happens rarely. Thus, for most cases, the need and overhead of accessing a separate ECC block do not occur.

Shafiee et al. [83] present a main memory compression approach, which uses rank subsetting approach. Their technique fetches compressed cache line from a single rank subset and the burst length depends on the compression factor. For example, an uncompressed and a highly compressed cache line can arrive with burst length of 64 and 8, respectively. Their proposed memory system stores the same number of blocks as the baseline and thus, they do not use compression to gain capacity advantage. Instead, the extra space provided by compression is used to implement stronger ECC or reduce energy by encoding the data in a format that reduces the number of data bus transitions.

5.6 Techniques for 3D memory systems

Park et al. [8] use compression along with power gating to save leakage energy in 3D stacked last level caches. Their technique tracks the frequency of access to each cache line. When a cache line has not been accessed for a given time, it is considered cold and hence, is compressed. The remaining lines are considered hot and are not compressed. Further, the unused cache portions are power-gated to save leakage energy. Thus, their technique aims to minimize the decompression overhead for frequently accessed cache lines, while also maximizing energy savings by compressing cold cache lines. Similar to Lee et al. [21], they also use a decompression buffer to minimize the decompression overhead.

Hybrid memory cube (HMC) design performs 3D stacking of DRAM on top of a logic die and uses through silicon vias (TSVs) as an interconnect between different layers. Khurshid et al. [12] note that high temperature variation may occur between different layers of HMC and use of throttling-based thermal management schemes can result in reduced performance and efficiency. They propose use of data compression to address this issue. In their technique, compression is performed in the on-chip memory controller and not in the logic die of HMC, to avoid the impact of energy overhead of compression on HMC. Reading/writing of compressed blocks requires fewer burst operations, which reduces the energy consumption and maximum temperature within the HMC. Also, compressed blocks are stored only in the hottest banks of the HMC to reduce the thermal gradient.

5.7 Techniques for non-volatile memories

Kong et al. [17] study the interaction of compression with redundant bit-write removal (RBR) to reduce writes in NVM memory. Both these approaches exploit data redundancy and hence, they observe that using compression with RBR provides nearly same reduction in write-traffic as RBR alone. This is expected, since the maximum write-traffic reduction is limited by redundancy present in data, which is already exploited by RBR and hence, use of compression does not provide extra benefits.

Dgien et al. [88] use compression to reduce bit-writes to NVM memory and also perform improved wear-leveling. Their technique uses FPC algorithm to compress data. The words which cannot be compressed are written in their uncompressed form. During a write operation, the new data bits are compared with the currently stored bits to only write the modified bits. On a read access, the data value is decompressed if it was stored in compressed form. Further, the additional space saved by compression is used to perform wear-leveling, such that the compressed data value is written to opposite sides of the word in the NVM array. This helps in achieving wear-leveling by uniformly spreading the writes to NVM (Such wear-leveling approach, in conjunction with compression is also used by Baek et al. [29]). The write-minimization and wear-leveling achieved by their technique leads to improved lifetime and reduction in write latency/energy.

Choi et al. [31] propose an adaptive compression technique for NVM cache which carefully controls additional writes due to compression. Their technique tracks the miss-rate of the compressed and uncompressed blocks by using set-sampling to determine whether cache compression is beneficial. Also, it tracks the number of write operations of the compressed and uncompressed blocks. If the miss rate of the uncompressed blocks is larger than that of compressed blocks, the incoming blocks are compressed. Otherwise, the number of writes to compressed and uncompressed blocks are compared and if the former is larger, then all incoming blocks are stored uncompressed.

Lee et al. [82] present a compression-based hybrid MLC/SLC (multi/single level cell) PCM management technique which brings the best of both MLC and SLC together. Their technique works on the following observations. A 2-bit MLC PCM can store two bits in a cell and thus provides higher capacity (density), however, its access latency is higher. By contrast, the SLC cell provides lower capacity but also provides nearly four times faster read/write access compared to MLC [82]. Their technique compresses data, and when the data can be compressed to less than 50%, it can fit in the same cell, but with SLC mode, thus providing higher performance. When the data cannot be compressed to less than 50%, it is stored in MLC mode at the cost of slow access. Their technique dynamically re-configures the PCM space into SLC or MLC modes and thus, does not require any profiling or address re-mapping.

Since PCM has low write-endurance and writes to it consume large latency/energy [30], researchers have proposed DRAM-PCM hybrid main memory where DRAM is used as a cache for PCM main memory. Du et al. [28] propose use of compression to increase the effective capacity of DRAM in hybrid memory systems. On a write access, their technique reads the existing data from PCM and computes the difference between the new data and existing data. Then, this difference-value is compressed and stored in DRAM. This approach converts unmodified data bits to zero before compression and hence, in general, provides high compression ratio. The trade-off in such delta compression approach is that in an attempt to reduce PCM writes, additional PCM reads are introduced. The DRAM itself is dynamically partitioned into a compressed and an uncompressed region. Only frequently-modified data are kept in the compressed region, while the performance-critical blocks are kept in uncompressed region to avoid the

latency of (de)compression.

Baek et al. [29] present a compression technique for DRAM-PCM hybrid main memory systems which works in two phases. In the first phase, data are compressed at the 'word-level' using an algorithm which computes difference between successive words of a block and stores only a single bit, if they are identical. This phase optimizes DRAM cache accesses and reduces the number of PCM accesses. In the second phase, compression is performed at 'bit-level' using FPC algorithm [40] which further reduces the number of PCM accesses.

5.8 Compression techniques for GPUs

Vijaykumar et al. [93] note that due to bottlenecks such as stalls due to main memory or data dependence, GPU resources remain underutilized. This provides opportunity to create hardware threads that accelerate the main (regular) threads by performing useful work. Their technique generates such threads and manages their execution at the granularity of a warp. The scheduling and execution of these 'assist warps' is done by hardware and allocation of resources (e.g. register file) to them is done by compiler. To alleviate memory bandwidth bottleneck, assist warps compress cache blocks before writing to memory and decompress them before placing them in caches and registers. Due to their diverse data patterns, different workloads benefit from different compression algorithms and hence, their technique allows implementing different algorithms (e.g. [35, 40, 49]) using different assist warps. Compared to hardware-only implementations of compression, their approach uses existing underutilized resources and does not require additional dedicated resources. Further, compression is used only for bandwidth-limited workloads (identified using static profiling), since compression is unbeneficial for compute-resource limited workloads and may even harm their performance due to overheads of assist warps.

GPUs use large register files (RFs) for holding concurrent execution contexts, this, however, also makes the power consumption of RFs a large fraction of GPU power consumption. Lee et al. [94] note that since all threads in a GPU warp execute the same instruction, register file (RF) access also happens at warp granularity. The threads in a warp are identified using thread-indices and hence, computations that depend on thread-indices operate on register data that show strong value similarity. Hence, within a warp, the arithmetic difference between two consecutive thread registers is small. Based on these observations, they present a warp-level register compression technique which uses BDI [35] algorithm to compress data. GPUs have banked RF organization and in their technique, a single register or a single register bank is selected as the base value for BDI algorithm and delta values are computed for remaining registers or banks. Compression of register values allows activating fewer register banks on each warp-level register access and this reduces the dynamic energy. Also, since the register content can be saved in fewer banks, unused banks can be power gated to save leakage energy. Thus, their technique saves GPU power.

5.9 Bandwidth compression techniques

Rogers et al. [4] study how bandwidth wall problem may restrict multicore scaling and to what extent can the bandwidth conservation techniques mitigate this challenge. They study a variety of techniques such as DRAM caches, use of smaller sized cores to enable larger sized caches, 3D stacked caches, cache compression and link compression etc. They assume that the die area is divided into some number of CEAs (core equivalent areas), where one CEA is equal to the area occupied by one processor core and its L1 caches. Assuming a baseline system with 8 cores and 8-CEA caches, they estimate the number of cores that can be supported in a next-generation processor with 32 total CEAs, under constraint that the memory traffic is kept constant relative to a baseline system. Without cache compression, only 11 cores can be supported. By comparison, for compression ratios of 1.3X, 2.0X and 3.0X, using cache compression alone can allow supporting 11, 13 and 14 cores, respectively and using both link and cache compression can allow supporting 13, 18 and 24 cores, respectively. This shows that compression techniques can be highly effective in permitting ongoing multicore scaling.

Alameldeen et al. [56] study the interaction of cache compression and prefetching and show that both these interact positively and can provide large performance improvements. They use cache compression and link compression to improve effective cache size, and reduces the miss rate and pin bandwidth requirement. They also propose adaptive prefetching mechanism that uses the extra tags employed by cache compression to detect useless and harmful prefetches. When the number of such prefetches increase more than a threshold, the prefetching is completely disabled.

Sathish et al. [89] present a technique for supporting lossless and lossy compression for data transferred through memory I/O links in GPUs. They note that for data values in several GPU workloads are highly compressible and since GPUs use 128B blocks, compared to 64B blocks in CPU, they can potentially provide higher compression ratio. In their technique, compression and decompression are performed in GPU's memory controller (MC), independent of the host CPU. The memory address space of GPU is partitioned into four address spaces, viz. global, texture, constant and local. Of these, their technique performs compression only on texture and global spaces, which

form majority of memory traffic. To avoid the complexity of space compaction, their technique does not use the space saved by the compression and thus, in their technique, compression does not increase memory capacity, but only reduces memory latency and bandwidth usage. A compressed 128B block can have a maximum of eight 16B chunks, of which only the compressed 16B chunk is fetched from the 128B block space. For several GPU workloads, reducing the precision of floating-point numbers harms the accuracy negligibly and hence, their technique does not fetch LSBs (least-significant bits) of FP data to save bandwidth and then applies lossless compression to the remaining data. Truncation is not performed for integer data, since it can harm the accuracy significantly.

5.10 Techniques used in real processors

Linux kernel provides zswap [90] as a form of virtual memory compression approach for storing swapped pages. When a memory page is to be swapped out, instead of moving it to a swap device (e.g. disk), zswap compresses the page and stores it in a memory pool dynamically allocated in system's RAM. Thus, zswap provides a compressed writeback cache functionality. Only when RAM runs out of space, the LRU (least recently used) page is evicted and moved to swap device. This defers and may even avoid writeback to the swap device, which reduces the I/O overhead of the system significantly.

IBM's active memory expansion (AME) technique [92] uses compression approach to expand the memory capacity of POWER7 systems. AME can reduce the physical memory requirement of logical partitions (LPARs) which allows consolidation of multiple logical partitions in a single system. AME can also be used to increase the effective memory capacity of an LPAR without increasing the physical memory used by it. AME can be selectively enabled for one or more LPARs on the system. Compression of a portion of LPAR's memory leads to formation of two pools, viz. a compressed pool and an uncompressed pool and their sizes are controlled based on the workload requirement.

Apple OS X provides "compressed memory" approach [91] which performs compression of inactive or least important applications to free the memory space. For compression, WKdm compression algorithm is used [38]. Compressed memory feature is especially useful for mobile products using SSD (solid state drive) storage where aftermarket expansion of RAM is challenging. Compared to the alternative of swapping data to disk which incurs significant processor and disk overheads, compression incurs smaller latency and reduces battery consumption by allowing processor and disk to power down more frequently. Compressed Memory feature can also function on top of virtual memory and can benefit from parallel processing on multiple cores to achieve high performance.

6 Conclusion and Future Challenges

In an era of data explosion, compression promises to be an effective approach for dealing with plateauing resource budgets. Moving forward, several challenges need to be addressed for fully leveraging the potential of compression approach in diverse range of computing systems and usage scenarios. In what follow, we discuss some of these challenges.

The efficacy of compression depends on data pattern and (de)compression steps also add significant overhead and complexity. Moreover, use of compression in a single processor component only, necessitates frequent (de)compression during data transfer. Integration of multiple compression approaches and use of compression in multiple processor components (e.g. cache, main memory, secondary storage) can prove effective in addressing these challenges.

With ongoing technological scaling, systems with large number of cores and large-sized LLCs are becoming commonly available. Conventional compression techniques which focus only on improving overall throughput and are oblivious of other system-level objectives and constraints, such as fairness, QoS (quality of service), implementing priorities, non-uniform latencies in large caches etc., will be inadequate for such systems. Design of novel compression approaches and their synergistic integration with other techniques will be an interesting research problem for computer architects.

As smart-phones outnumber the population of earth and become used for even more data-intensive applications (such as multimedia), use of compression approach in smart-phones has become, not only attractive, but even imperative. However, smart-phones also impose strict power and cost budgets. Evidently, computer architects need to develop highly efficient compression techniques for justifying their use in such portable systems.

In this paper, we presented a survey of approaches for utilizing data compression in cache and main memory systems. We classified the techniques based on several important characteristics to highlight their similarities and differences. It is hoped that this survey will provide the researchers valuable insights into compression approach and its potential role in memory systems of future computing systems.

References

[1] S. Mittal, "Power management techniques for data centers: A survey," Oak Ridge National Laboratory, USA, Tech. Rep. ORNL/TM-2014/381, 2014.

[2] B. Giridhar, M. Cieslak, D. Duggal, R. Dreslinski, H. M. Chen, R. Patti, B. Hold, C. Chakrabarti, T. Mudge, and D. Blaauw,

"Exploring DRAM organizations for energy-efficient and resilient exascale memories," in *SC*, 2013, p. 23.

[3] J. S. Vetter and S. Mittal, "Opportunities for nonvolatile memory systems in extreme-scale high performance computing," *Computing in Science and Engineering, special issue*, 2015.

[4] B. M. Rogers *et al.*, "Scaling the bandwidth wall: challenges in and avenues for CMP scaling," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, 2009, pp. 371–382.

[5] S. Mittal, "A survey of architectural techniques for improving cache power efficiency," *Elsevier Sustainable Computing: Informatics and Systems*, vol. 4, no. 1, pp. 33–43, 2014.

[6] S. Roy, R. Kumar, and M. Prvulovic, "Improving system performance with compressed memory," in *International Parallel and Distributed Processing Symposium*, 2001, pp. 7–13.

[7] S. Biswas, T. Carley, M. Simpson, B. Middha, and R. Barua, "Memory overflow protection for embedded systems using run-time checks, reuse, and compression," *ACM TECS*, vol. 5, no. 4, pp. 719–752, 2006.

[8] J. Park, J. Jung, K. Yi, and C.-M. Kyung, "Static energy minimization of 3D stacked L2 cache with selective cache compression," in *International Conference on Very Large Scale Integration (VLSI-SoC)*, 2013, pp. 228–233.

[9] O. Ozturk and M. Kandemir, "ILP-Based energy minimization techniques for banked memories," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 3, p. 50, 2008.

[10] K. Tanaka and A. Matsuda, "Static energy reduction in cache memories using data compression," in *IEEE TENCON*, 2006, pp. 1–4.

[11] S. Mittal, "A Survey of Architectural Techniques For DRAM Power Management," *International Journal of High Performance Systems Architecture*, vol. 4, no. 2, pp. 110–119, 2012.

[12] M. J. Khurshid and M. Lipasti, "Data compression for thermal mitigation in the Hybrid Memory Cube," in *International Conference on Computer Design (ICCD)*, 2013, pp. 185–192.

[13] http://www.green500.org/, 2015.

[14] Y. Tian, S. M. Khan, D. A. Jiménez, and G. H. Loh, "Last-level cache deduplication," in *ACM international conference on Supercomputing*, 2014, pp. 53–62.

[15] A. Arelakis and P. Stenstrom, "A case for a value-aware cache," *Computer Architecture Letters*, vol. 13, no. 1, pp. 1–4, 2014.

[16] A. Arelakis and P. Stenstrom, "SC2: a statistical compression cache scheme," in *International Symposium on Computer Architecuture*, 2014, pp. 145–156.

[17] J. Kong and H. Zhou, "Improving privacy and lifetime of PCM-based main memory," in *International Conference on Dependable Systems and Networks (DSN)*, 2010, pp. 333–342.

[18] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *ISCA*, 2004, pp. 212–223.

[19] E. G. Hallnor and S. K. Reinhardt, "A unified compressed memory hierarchy," in *HPCA*, 2005, pp. 201–212.

[20] F. Douglis, "The compression cache: Using on-line compression to extend physical memory." in *USENIX Winter Conference*, 1993, pp. 519–529.

[21] J.-S. Lee, W.-K. Hong, and S.-D. Kim, "Design and evaluation of a selective compressed memory system," in *International Conference on Computer Design (ICCD)*, 1999, pp. 184–191.

[22] B. Abali, H. Franke, D. E. Poff, R. Saccone, C. O. Schulz, L. M. Herger, and T. B. Smith, "Memory expansion technology (MXT): software support and performance," *IBM Journal of Research and Development*, vol. 45, no. 2, pp. 287–301, 2001.

[23] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *MICRO*, 2013, pp. 172–184.

[24] N. Kim, T. Austin, and T. Mudge, "Low-energy data cache using sign compression and cache line bisection," in *2nd Annual Workshop on Memory Performance Issues (WMPI)*, 2002.

[25] J.-S. Lee, W.-K. Hong, and S.-D. Kim, "An on-chip cache compression technique to reduce decompression overhead and design complexity," *Journal of systems Architecture*, vol. 46, no. 15, pp. 1365–1382, 2000.

[26] S. Mittal, J. S. Vetter, and D. Li, "A Survey Of Architectural Approaches for Managing Embedded DRAM and Non-volatile On-chip Caches," *IEEE TPDS*, 2014.

[27] P. Pujara and A. Aggarwal, "Restrictive compression techniques to increase level 1 cache capacity," in *International Conference on Computer Design (ICCD)*, 2005, pp. 327–333.

[28] Y. Du, M. Zhou, B. Childers, R. Melhem, and D. Mossé, "Delta-compressed caching for overcoming the write bandwidth limitation of hybrid main memory," *ACM TACO*, vol. 9, no. 4, p. 55, 2013.

[29] S. Baek, H. G. Lee, C. Nicopoulos, and J. Kim, "A dual-phase compression mechanism for hybrid DRAM/PCM main memory architectures," in *GLSVLSI*, 2012, pp. 345–350.

[30] S. Mittal, "A survey of power management techniques for phase change memory," *International Journal of Computer Aided Engineering and Technology (IJCAET)*, 2015.

[31] J. H. Choi, J. W. Kwak, S. T. Jhang, and C. S. Jhon, "Adaptive cache compression for non-volatile memories in embedded system," in *Conference on Research in Adaptive and Convergent Systems*. ACM, 2014, pp. 52–57.

[32] M. N. Bojnordi and E. Ipek, "DESC: energy-efficient data exchange using synchronized counters," in *International Symposium on Microarchitecture*, 2013, pp. 234–246.

[33] S. Mittal, Z. Zhang, and J. Vetter, "FlexiWay: A Cache Energy Saving Technique Using Fine-grained Cache Reconfiguration," in *International Conference on Computer Design (ICCD)*, 2013.

[34] D. Chen, E. Peserico, and L. Rudolph, "A dynamically partitionable compressed cache," *Proc. the Singapore-MIT Alliance Symposium*, 2003.

[35] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: practical data compression for on-chip caches," in *PACT*, 2012, pp. 377–388.

[36] L. Yang, R. P. Dick, H. Lekatsas, and S. Chakradhar, "Online memory compression for embedded systems," *ACM Transactions on Embedded Computing Systems*, vol. 9, no. 3, p. 27, 2010.

[37] D. Nakar and S. Weiss, "Selective main memory compression by identifying program phase changes," in *3rd workshop on Memory performance issues*, 2004, pp. 96–101.

[38] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis, "The case for compressed caching in virtual memory systems," in *USENIX Annual Technical Conference, General Track*, 1999, pp. 101–116.

[39] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[40] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for L2 caches," Univ. Wisconsin-Madison, Tech. Rep., 2004.

[41] L. Villa, M. Zhang, and K. Asanović, "Dynamic zero compression for cache energy reduction," in *International Symposium on Microarchitecture*, 2000, pp. 214–220.

[42] I. Chihaia and T. Gross, "An analytical model for software-only main memory compression," in *Workshop on Memory performance issues*, 2004, pp. 107–113.

[43] Y. Zhang and R. Gupta, "Data compression transformations for dynamically allocated data structures," in *Compiler Construction*. Springer, 2002, pp. 14–28.

[44] O. Ozturk, M. Kandemir, and G. Chen, "Access pattern-based code compression for memory-constrained systems," *ACM TODAES*, vol. 13, no. 4, p. 60, 2008.

[45] S. Tang, H. Tan, and L. Li, "A novel memory compression technique for embedded system," in *International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)*, 2012, pp. 287–292.

[46] L. Benini, D. Bruni, A. Macii, and E. Macii, "Memory energy minimization by data compression: algorithms, architectures and implementation," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 12, no. 3, pp. 255–268, 2004.

[47] H. Lekatsas and W. Wolf, "SAMC: a code compression algorithm for embedded processors," *IEEE TCAD*, vol. 18, no. 12, pp. 1689–1701, 1999.

[48] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Linearly compressed pages: A main memory compression framework with low complexity and low latency," CMU, Tech. Rep. 2012-005, 2012.

[49] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm," *IEEE Transactions on VLSI Systems*, vol. 18, no. 8, pp. 1196–1208, 2010.

[50] S. Sardashti and D. A. Wood, "Decoupled compressed cache: exploiting spatial locality for energy-optimized compressed

caching," in *MICRO*, 2013, pp. 62–73.
[51] L. Chen, Y. Cao, and Z. Zhang, "Free ECC: An efficient error protection for compressed last-level caches," in *ICCD*. IEEE, 2013, pp. 278–285.
[52] S. Kim, J. Lee, J. Kim, and S. Hong, "Residue cache: a low-energy low-area L2 cache architecture via compression and partial hits," in *MICRO*, 2011, pp. 420–429.
[53] Y. Xie and G. H. Loh, "Thread-aware dynamic shared cache compression in multi-core processors," in *ICCD*. IEEE, 2011, pp. 135–141.
[54] S. Baek, H. G. Lee, C. Nicopoulos, J. Lee, and J. Kim, "ECM: Effective Capacity Maximizer for high-performance compressed caching," in *HPCA*, 2013, pp. 131–142.
[55] G. Pekhimenko, T. Huberty, R. Cai, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Exploiting compressed block size as an indicator of future reuse," CMU, Tech. Rep. TR-SAFARI-2013-003, 2013.
[56] A. R. Alameldeen and D. A. Wood, "Interactions between compression and prefetching in chip multiprocessors," in *HPCA*, 2007, pp. 228–239.
[57] J. Dusser, T. Piquet, and A. Seznec, "Zero-content augmented caches," in *ICS*, 2009, pp. 46–55.
[58] A.-R. Adl-Tabatabai, A. M. Ghuloum, and S. O. Kanaujia, "Compression in cache design," in *International conference on Supercomputing*, 2007, pp. 190–201.
[59] G. Keramidas, K. Aisopos, and S. Kaxiras, "Dynamic dictionary-based data compression for level-1 caches," in *Architecture of Computing Systems (ARCS)*, 2006, pp. 114–129.
[60] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *MICRO*, 2000, pp. 258–265.
[61] Y. Zhang and R. Gupta, "Enabling partial cache line prefetching through data compression," in *International Conference on Parallel Processing*, 2003, pp. 277–285.
[62] S. Sardashti, A. Seznec, and D. A. Wood, "Skewed Compressed Caches," in *MICRO*, 2014.
[63] G. Pekhimenko, T. Huberty, R. Cai, O. Mutlu, P. P. Gibbons, M. A. Kozuch, and T. C. Mowry, "Exploiting Compressed Block Size as an Indicator of Future Reuse," in *HPCA*, 2015.
[64] P.-Y. Hsu, P.-L. Lin, and T. Hwang, "Compaction-free compressed cache for high performance multi-core system," in *ICCAD*, 2014, pp. 140–147.
[65] A. Vishnoi, P. R. Panda, and M. Balakrishnan, "Cache aware compression for processor debug support," in *Conference on Design, Automation and Test in Europe*, 2009, pp. 208–213.
[66] R. Das, A. K. Mishra, C. Nicopoulos, D. Park, V. Narayanan, R. Iyer, M. S. Yousif, and C. R. Das, "Performance and power optimization through data compression in network-on-chip architectures," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2008, pp. 215–225.
[67] C. Nitta and M. Farrens, "Techniques for increasing effective data bandwidth," in *International Conference on Computer Design (ICCD)*, 2008, pp. 514–519.
[68] N. R. Mahapatra, J. Liu, and K. Sundaresan, "The performance advantage of applying compression to the memory system," in *ACM SIGPLAN Notices*, vol. 38, no. 2, 2002, pp. 86–96.
[69] K. Shrivastava and P. Mishra, "Dual code compression for embedded systems," in *International Conference on VLSI Design (VLSI Design)*, 2011, pp. 177–182.
[70] Y. Zhang, J. Yang, and R. Gupta, "Frequent value locality and value-centric data cache design," in *ACM SIGOPS Operating Systems Review*, vol. 34, no. 5, 2000, pp. 150–159.
[71] V. Phalke and B. Gopinath, "Compression-based program characterization for improving cache memory performance," *IEEE TC*, vol. 46, no. 11, pp. 1174–1186, 1997.
[72] P. Petrov and A. Orailoglu, "Tag compression for low power in dynamically customizable embedded processors," *IEEE TCAD*, vol. 23, no. 7, pp. 1031–1047, 2004.
[73] C. Lattner and V. S. Adve, "Transparent pointer compression for linked data structures," in *Workshop on Memory system performance*. ACM, 2005, pp. 24–35.
[74] R. S. De Castro, A. Lago, and M. Silva, "Adaptive compressed caching: Design and implementation," in *SBAC-PAD*. IEEE, 2003, pp. 10–18.
[75] I. C. Tuduce and T. R. Gross, "Adaptive main memory compression," in *USENIX Annual Technical Conference*, 2005, pp. 237–250.
[76] V. Beltran, J. Torres, and E. Ayguadé, "Improving web server performance through main memory compression," in *ICPADS*. IEEE, 2008, pp. 303–310.
[77] V. Beltran, J. Torres, and E. Ayguadé, "Improving disk bandwidth-bound applications through main memory compression," in *workshop on MEmory performance: DEaling with Applications, systems and architecture*. ACM, 2007, pp. 57–63.
[78] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2, 2005, pp. 74–85.
[79] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan, "Live virtual machine migration with adaptive, memory compression," in *IEEE International Conference on Cluster Computing and Workshops (CLUSTER)*, 2009, pp. 1–10.
[80] L. R. Vittanala and M. Chaudhuri, "Integrating memory compression and decompression with coherence protocols in distributed shared memory multiprocessors," in *International Conference on Parallel Processing (ICPP)*, 2007, pp. 4–4.
[81] J. Dusser and A. Seznec, "Decoupled zero-compressed memory," in *HiPEAC*, 2011, pp. 77–86.
[82] H. G. Lee, S. Baek, J. Kim, and C. Nicopoulos, "A Compression-Based Hybrid MLC/SLC Management Technique for Phase-Change Memory Systems," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2012, pp. 386–391.
[83] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, "Memzip: Exploring unconventional benefits from memory compression," in *HPCA*, 2014.
[84] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Wolf, "Energy savings through compression in embedded java environments," in *International symposium on Hardware/software codesign*, 2002, pp. 163–168.
[85] H. Hajimiri, K. Rahmani, and P. Mishra, "Synergistic integration of dynamic cache reconfiguration and code compression in embedded systems," in *International Green Computing Conference and Workshops (IGCC)*, 2011, pp. 1–8.
[86] S.-W. Seong and P. Mishra, "Bitmask-based code compression for embedded systems," *IEEE TCAD*, vol. 27, no. 4, pp. 673–685, 2008.
[87] M. Kjelso, M. Gooch, and S. Jones, "Design and performance of a main memory hardware data compressor," in *EUROMICRO*, 1996, pp. 423–430.
[88] D. B. Dgien, P. M. Palangappa, N. A. Hunter, J. Li, and K. Mohanram, "Compression architecture for bit-write reduction in non-volatile memory technologies," in *International Symposium on Nanoscale Architectures (NANOARCH)*, 2014, pp. 51–56.
[89] V. Sathish, M. J. Schulte, and N. S. Kim, "Lossless and lossy memory i/o link compression for improving performance of gpgpu workloads," in *PACT*, 2012, pp. 325–334.
[90] S. Jennings, "The zswap compressed swap cache," https://lwn.net/Articles/537422/, 2013.
[91] D. E. Dilger, http://goo.gl/ucQpFO, 2013.
[92] "Active Memory Expansion: Overview and Usage Guide," http://goo.gl/4qe5wS, 2010.
[93] N. Vijaykumar *et al.*, "A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps," in *ISCA*, 2015.
[94] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, " Warped-Compression: Enabling Power Efficient GPUs through Register Compression," in *ISCA*, 2015.

Sparsh Mittal received the B.Tech. degree in electronics and communications engineering from IIT, Roorkee, India and the Ph.D. degree in computer engineering from Iowa State University, USA. He is currently working as a Post-Doctoral Research Associate at ORNL. His research interests include non-volatile memory, memory system power efficiency, cache and GPU architectures.

Jeffrey S. Vetter received the Ph.D. from Georgia Institute of Technology (GT). He holds a joint appointment between ORNL and GT. At ORNL, he is a Distinguished R&D Staff Member, and the founding group leader of the Future Technologies Group. At GT, he is a Joint Professor in the Computational Science and Engineering School, the Project Director for the NSF Track 2D Experimental Computing Facility for large scale heterogeneous computing using graphics processors, and the Director of the NVIDIA CUDA Center of Excellence. His research interests include massively multithreaded processors, non-volatile memory and heterogeneous multicore processors.