HyComp: A Hybrid Cache Compression Method for Selection of Data-Type-Specific Compression Methods

Angelos Arelakis*‡
angelos@chalmers.se

Fredrik Dahlgren*†
fredrik.dahlgren@ericsson.com

Per Stenstrom*‡
per.stenstrom@chalmers.se

*Chalmers University of Technology
Göteborg, Sweden

‡ZeroPoint Technologies AB
Göteborg, Sweden

†Ericsson AB
Lund, Sweden

ABSTRACT

Proposed cache compression schemes make design-time assumptions on value locality to reduce decompression latency. For example, some schemes assume that common values are spatially close whereas other schemes assume that null blocks are common. Most schemes, however, assume that value locality is best exploited by fixed-size data types (e.g., 32-bit integers). This assumption falls short when other data types, such as floating-point numbers, are common. This paper makes two contributions. First, HyComp – a hybrid cache compression scheme – selects the best-performing compression scheme, based on heuristics that predict data types. Data types considered are pointers, integers, floating-point numbers and the special (and trivial) case of null blocks. Second, this paper contributes with a compression method that exploits value locality in data types with predefined semantic value fields, e.g., as in the exponent and the mantissa in floating-point numbers. We show that HyComp, augmented with the proposed floating-point-number compression method, offers superior performance in comparison with prior art.

Categories and Subject Descriptors

B.3.2 [**Design Styles**]: Cache memories; E.4 [**Coding and information theory**]: Data compaction and compression

Keywords

Cache Compression, Huffman Coding, Hybrid Compression, Floating-Point Data, Value Locality

1. INTRODUCTION

Large last-level caches (LLC) help reduce the speed gap between processing and off-chip memory access. To this end, enlarging the LLC space requires to sacrifice processing cores in the chip's real estate and may result in longer cache access latencies and more energy consumption. Cache compression [1, 2, 3, 4, 5, 6] is a promising approach to increase cache capacity by exploiting the available cache resources more efficiently.

Cache compression has, however, disadvantages too: Decompression inevitably adds to the cache access latency. For this reason, proposed compression methods make design-time assumptions about the most common values to simplify the decompression process and keep decompression latency low. For example, Frequent Pattern Compression (FPC) [2] assumes that the most frequent values are either the value zero or narrow integers. On the other hand, Base-Delta-Immediate compression (BDI) [3] assumes that common values are spatially close to each other making it meaningful to encode them with their difference to a base value. In contrast, Zero-Content Augmented Cache (ZCA) [1] assumes that null blocks, i.e., blocks filled with the value zero, are common and encode these with a single bit. Finally, SC^2 [5] Huffman-encodes fixed-size values (32-bit integers) based on value frequency. While SC^2 [5] appears to be a reasonable default strategy, it does not always result in the highest compressibility. For example, a null block is more compactly encoded using ZCA than SC^2. Moreover, integers that are moderately common are in SC^2 replaced by longer code-words than the most common ones. Therefore, if these integers are also spatially close, they can potentially be coded much denser by BDI instead. In essence, none of the proposed schemes are hitherto always better than others.

A second problem with compression methods so far is their design-time assumption that fixed-size data types (e.g., 32-bit integers) are the root cause of value locality. This assumption falls short in light of, e.g., floating-point numbers according to IEEE-754 or other data types such as text strings. What is needed is a compression scheme that can exploit data locality exhibited by data types with predefined semantic fields, such as exponents and mantissas in floating-point numbers.

This paper makes two major contributions: First, it presents *HyComp*, a hybrid compression method tailored for caches that selects the best compression method based on data-type prediction. Data-type prediction has been also used to improve link compression schemes [7,

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.
Copyright is held by the owner/author(s).
MICRO-48, December 05-09, 2015, Waikiki, HI, USA
ACM 978-1-4503-4034-2/15/12
http://dx.doi.org/10.1145/2830772.2830823.

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 14:12:56 UTC from IEEE Xplore. Restrictions apply.

8] but not to select the best performing compression method among a number of different ones. We show that HyComp can, at run-time, select between integers, pointers, floating-point numbers as well as the trivial special case of null blocks. The paper shows that HyComp can quickly and with an accuracy that often exceeds 80%, select between example data-type-specific methods expected to show the best compressibility, such as ZCA, BDI, SC^2 and a compression method tailored to floating-point data.

Secondly, this paper presents a compression method that exploits value locality in data types with predefined semantic value fields, such as the exponent and mantissa in floating-point numbers. While it is well known [9, 10, 11] that there is significant value locality in the exponent, we show for the first time, that there is significant value locality in the most significant bits of the mantissa as well. Our proposed compression method, referred to as FP-H, exploits this insight and we show that significant compressibility can be achieved in floating-point intensive applications.

We have incorporated HyComp and FP-H in an architectural model of an LLC in a multi-core system. We study in detail the impact of the compression / decompression engines on performance and energy consumption and show that it provides superior compressibility and performance gains in comparison with prior work.

As for the rest of the paper, Section 2 provides motivational data for the case of hybrid cache compression schemes and for data-type-aware compression methods. Then, Sections 3 and 4 describe our proposed compression methods: HyComp and our data-type aware floating-point scheme, respectively. Sections 5 and 6 present our methodology and results, respectively. We end the paper by positioning our contributions in relation to prior work after which we conclude.

2. MOTIVATION

Prior cache compression schemes target different sets of common values but have in common that value locality stems from 32-bit entities. Floating-point numbers constitute one example data type where this assumption falls short. Figure 1 shows the compressibility of a 1-MB cache for different compression algorithms across 14 workloads after removing the null blocks. We remove the null blocks because, as shown later, these can be effectively compressed by e.g. ZCA [1]. The 7 first workloads are realistic data sets from the Florida Sparse Matrix Collection [12]; lu, mg and is are from NAS [13]; cactus and gromacs are from SPEC2006FP and the rest are SPEC2006INT applications. We provide results for the entire SPEC2006 suite in Section 6.

BDI [3], SC^2 [5] and C-PACK [4] represent state of the art cache compression schemes. FPC [14] is a state-of-the-art method in IEEE double-precision floating-point compression proposed by Burtscher and Ratanaworabhan. FP-H is our proposed data-type aware cache compression scheme for floating-point (FP) data and Hybrid is our proposed HyComp scheme. FPC can only compress streams of data; because of this, it is not suitable for cache compression, but it is used for reference. LZMA[1] is an efficient software implemented dictionary-based compression algorithm (used by 7-Zip). While LZMA is used as a reference, because of its high compression ratio, it demands complex and slow software algorithms and is therefore not applicable to caches that demand fast decompression.

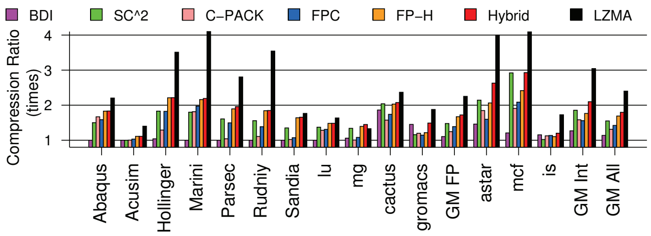

Figure 1: Compressibilty of HyComp and FP-H in relation to prior art.

First, BDI assumes that value locality stems from values that are spatially close. For example, addresses that differ by a small amount will be encoded densely by the difference to a base address. In contrast, SC^2 uses Huffman coding to encode frequent values densely and, like BDI, assumes that data types expose value locality using fixed-size entities (e.g., 32 bits). On the other hand, C-PACK compresses based on patterns and a small dictionary looking for small and repeated values still in 32-bit granularities.

From Figure 1, we can make several important observations. The LZMA bar shows that there is significant value locality in the set of workloads after null blocks have been removed. However, it is challenging to uncover this level of compressibility with low decompression latency. BDI, SC^2 and C-PACK have been shown to support fast compression/decompression, but their compressibility is quite limited. Considering first their average (geometric mean) compressibility across workloads, Figure 1 shows that BDI and C-PACK offer a compressibility of 1.3X or less. While SC^2 shows substantially higher compressibility, especially for integer applications, the average compressibility is limited to about 1.5X. BDI shows higher compression in gromacs where there are many pointer values.

The limited compressibility of previously proposed methods is mainly attributed to FP intensive applications. From Figure 1, we can see that BDI and C-PACK perform, in general, poorly for FP intensive applications, whereas SC^2 exploits some value locality. For cactus from the SPEC2006FP suite, compressibility results are mixed for BDI, C-PACK and SC^2 but in general below 2X.

The main limitation of previously proposed compression schemes is that they make a priori assumptions on what data types are the root cause of value locality. First, if all data were FP numbers, the way the exponent and mantissa are represented would be explicit making it interesting to unlock the value locality

[1]Lempel-Ziv-Markov chain

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 14:12:56 UTC from IEEE Xplore. Restrictions apply.

in each of these entities. Our proposed FP-H scheme does exactly that and Figure 1 shows how it performs. We can see that for FP workloads, FP-H does better than BDI, C-PACK and SC^2 in all cases, except for gromacs. It also compresses better than FPC, which unlike FP-H, does not break the 64-bit (double precision) FP value in its fields but rather tries to predict it based on history, using also shift operations to eliminate the irregular mantissa's least-significant bits.

Unfortunately, memory or cache content does not carry any semantic information about data types. HyComp approaches this issue by predicting the data type based on characteristics of 64-bit entities to distinguish between integers, pointers and FP numbers. As we can see in Figure 1, HyComp outperforms all prior schemes by selecting the best compression method based on its prediction. Section 3 describes HyComp whereas Section 4 describes FP-H.

3. HYCOMP: HYBRID COMPRESSION

3.1 Overview

Our baseline system is a multi-core architecture with a multi-level cache hierarchy. HyComp is integrated in the last-level cache (LLC) as in prior approaches [5, 15, 16]. Figure 2 illustrates how HyComp is integrated in a conventional LLC with a number of known compression methods. Although data could be compressed in memory and caches closer to the processor, called upper-level caches, we assume in this paper that data are only compressed in the LLC. Hence, data are compressed before inserted into the LLC from the memory, or when they are written back to LLC from an upper-level cache. On the other hand, when a block is evicted from LLC, or when it is fetched by an upper-level cache it is decompressed. While the compression latency can be hidden, the decompression latency is on the critical memory access path and may lead to performance loss.

HyComp compresses blocks inserted from memory or blocks written back from the upper-level cache by making a data-type prediction using heuristics. Based on the prediction, it selects among a set of compression methods the one expected to provide the best compressibility. The compressed block is then inserted in the LLC and the type of method selected is recorded as metadata in the tag store. Conversely, HyComp simply decompresses a block by selecting one out of a set of decompression methods based on the metadata in the tag store. Thus, while compression will take longer time, decompression will be as fast as using a single compression method. We next describe our assumed association between common data types and compression methods that the HyComp embodiment evaluated in this paper leverages upon.

3.2 Data Types vs. Compression Methods

While the general idea behind HyComp can potentially be built on any set of available compression methods, we have selected methods that perform well for certain common data types. The data types considered

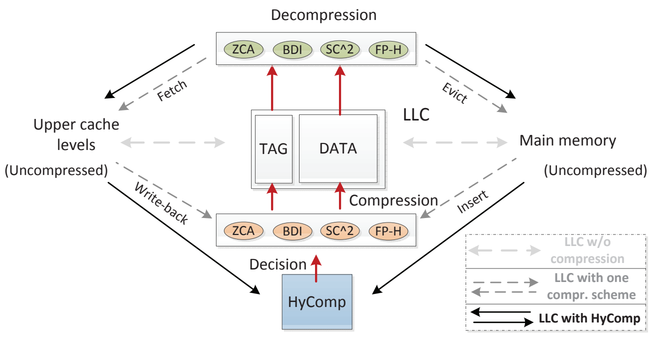

Figure 2: HyComp integration in the LLC.

in this paper are integers, pointers and 64-bit floating-point numbers. In addition, while not a data type per se, we also consider the trivial case when all 64-bit values in a cache block are zero, referred to as null blocks. The particular HyComp embodiment evaluated in this paper associates 64-bit data types with compression methods as follows:

- **Integers:** Integers are associated with SC^2 [5] because it assigns denser codes to more frequently used integers than other schemes by using Huffman coding [17].

- **Pointers:** As pointers typically exhibit spatial value locality, meaning that values differ by a small amount, they are associated with BDI [3]. The limitation of BDI in its main configuration with two bases (one as zero) is that a block is compressed only, if all the values contained in the block fall in two ranges; otherwise it is left uncompressed. This is taken into account when building HyComp.

- **Floating-point numbers (FP):** FP numbers are associated with FP-H, a novel compression method proposed in this paper and described in Section 4.

- **Null blocks:** Null blocks are associated with Zero-content augmented cache compression (ZCA) [1]. Null blocks are common [18, 19]. We encode null blocks with a single bit. A block can be considered null even when it stores negative zero FP values. We refer to this as *negative null block*.

3.3 Heuristics for Prediction of Data Types

When compressing a block, one could aggressively try each of the aforementioned compression methods, one-by-one, and then select the one with the best compression ratio. We refer to this as the *brute-force mode*. The advantage of this approach is optimality in terms of compressibility, but it could lead to prohibitive power consumption and possibly prohibitive compression latency. However, as a point of comparison, we use the brute-force mode in our evaluations to establish the upper-bound potential of HyComp.

Pragmatically, HyComp uses instead heuristics to predict the data type and then apply the compression method associated with it, as defined in the previous section.

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 14:12:56 UTC from IEEE Xplore. Restrictions apply.

Conceptually, within each block (say 64 bytes), it applies its heuristics to each 64-bit (8-byte) entity, called a *chunk*. The data type of each chunk is first classified. HyComp then counts the occurrences of each predicted data type within the block and predicts a predominant data type that is associated with a single compression method. That compression method is applied to all chunks, i.e., 64-bit entities, in the block.

HyComp compresses a block in two process steps: **Phase-I**: The block is divided into chunks. For each chunk, it inspects particular bit-portions referred to as *Inspection Portions* (IPs) and classifies the data type of the chunk. **Phase-II**: Based on the classification of the individual chunks, a best compression method is predicted and selected for the block.

In Phase-I, the following data-type classifications are performed in parallel. Note: The IP size depends on the data types being predicted:

- **Integers:** The IP is the 4 most significant bytes of a chunk. If IP is either 0x00000000 (i.e., positive integer) or 0xFFFFFFFF (i.e., negative integer), the chunk is classified as an integer. In addition, a flag is set if a negative value encoded in a chunk is detected - the treatment of that flag is described later. If a chunk is classified as an integer, a counter ($\#Int$) is incremented.

- **Pointers:** The IP is the 4 most significant bytes of a chunk. If the two most significant bytes of the IP =0x0000 and the two least significant bytes of the IP $\neq$0x0000, the chunk is classified as a pointer. The rationale is to neither classify small integers nor small pointers as pointers. A counter ($\#Ptr$) is incremented for each chunk classified as pointer.

- **Floating-point numbers (FP):** The IP is the 7 bits (part of exponent) placed next to the most significant bit (the sign bit) of a chunk, as further justified in Section 4 where FP-H is described. The rationale is that FP numbers contained in a block often have the same exponents or their exponents are clustered (the 7-bit IP). HyComp makes pairwise comparisons between adjacent and alternative IPs, and increments $\#FP$ when they match.

- **(Negative) Null blocks:** The IP is a chunk. If all chunks have the null value (0x00...0) or a negative zero value (0x80...0), the block is classified as a null block.

In Phase-II, the heuristic selects the best compression method for the block by evaluating the classifications carried out in Phase-I in a certain order, as depicted in Figure 3. As a first step, if a block is classified as a (negative) null block, ZCA is selected. Then it checks whether all counts (#Int, #FPs and #Ptr) are zero. This is expected to be rare, but if it holds HyComp speculates that data will not compress efficiently because of randomness, being encrypted or belonging to a data type not considered. The next step is to evaluate the flag and $\#Int$ for integer prediction to select SC^2.

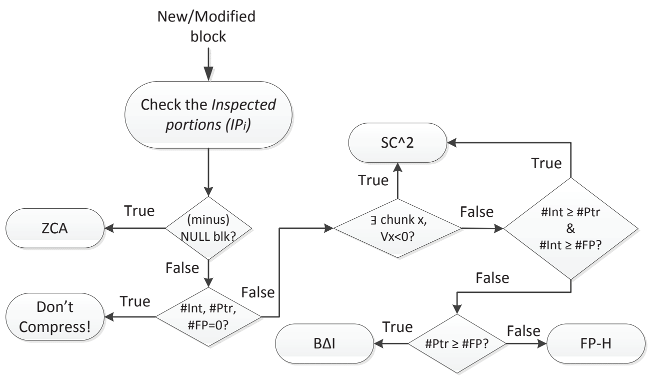

Figure 3: HyComp classification process.

After this point in the process, the selected compression method is based on the count of $\#Ptr$ and $\#FPs$. The maximum count selects the compression method according to the association between data types and compression methods described in Section 3.2.

Finally, this heuristic is fully implemented in hardware to accelerate the decisions. The characterization phase (Phase-I) requires inspecting various bit portions and increment counters. Each characterization is performed in parallel using bitwise comparators and adders. In Phase-II, the decision is made using priority encoders. The detailed delays and overheads for HyComp are described in Section 5.2.

3.4 HyComp Cache Design

To support cache compression using any compression method, a conventional cache must be re-designed to pack multiple compressed blocks into a cache-block/set frame. Our baseline does this as follows. First, the tag store is decoupled from the data store [6, 5]. To accommodate a compression ratio of K, K tags are associated with each block frame as shown on the right of Figure 4, where K=2. While this creates tag-store overhead, it also enables aggressive compression [5]. Second, to use the block frame space efficiently, we allow a compressed block to be placed at an arbitrary position inside the frame. To accommodate this, we adopt the methodology of SC^2 [5] to use K pointers per block frame to point to the exact position in the data store, where a compressed block is stored. We evaluate later a design where the resolution is a single byte.

Finally, fragmentation is tackled by adopting the same methodology for free-space management, as proposed in SC^2 [5]. Specifically, if a block being written back from an upper-level cache to the LLC is compressed to a larger size than the one selected for eviction in the LLC, more blocks in the LLC need to be evicted. Evicting the next LRU blocks may take time, as compaction is also required. According to that methodology and as shown on the bottom of Figure 4, adjacent blocks are evicted but only if they are not null, otherwise no space is gained unless we run out of tags [3]. On the other hand, if the block is smaller than the old one, compaction is done in order to release more contiguous space to the next LRU block. This happens, however, in

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 14:12:56 UTC from IEEE Xplore. Restrictions apply.

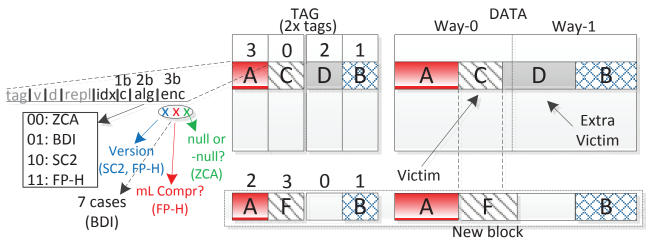

Figure 4: HyComp cache structure.

the background after the block has been written. Compaction is off of the critical access path.

Beyond the typical control bits (tag, valid, dirty and replacement bits – shown in grey on the left of Figure 4) and the index (i.e., pointer to the block frame), HyComp adds the following status bits in the tag store associated with a block frame: Compression status (*c* – 1-bit), Method (*alg* – 2-bits) and Special encoding (*enc* – 3-bits). The c bit indicates whether a block is compressed. The used compression scheme is recorded by the method bits, as shown in Figure 4, while the special encoding (enc.) is used in different ways depending on the selected method:

- BDI: The special encoding determines the base/delta granularity used: 1) Base=8 bytes and Delta=1, 2 or 4 bytes (3 cases); 2) Base=4 bytes and Delta is 1 or 2 bytes (2 cases); 3) 2) Base=2 bytes and Delta is 1; and 4) Base is 8 bytes and delta is 0 (i.e., compression of repeated values).

- ZCA: The rightmost bit of the special encoding determines whether it is a null block or a negative null block. The rest of the bits are not used.

- SC^2 or FP-H: As two Huffman encodings may exist for each scheme at the same time in order to allow smooth transitions from an old encoding to a new one [5], the leftmost bit determines the right Huffman-encoding's version. The middle bit is used only by FP-H to determine whether the group of Mantissa-Low subfields is uncompressed (see Section 4). The leftmost bit is not used when these methods are selected.

The area overheads are presented in Section 5. Moreover, since value locality changes slowly [5], SC^2 and FP-H create their Huffman encodings in software assuming that encodings change once every second.

4. FP-H: SEMANTIC BIT-FIELD COMPRESSION APPLIED TO FP NUMBERS

Previous work [9, 10] notes that value locality is only exposed in the exponent. By contrast, in this paper, FP-H exploits the observation that there is ample value locality also in the mantissa, if it is partitioned. To exploit this, FP-H divides a floating-point number into three fields: Exponent, Mantissa-High and Mantissa-Low and employs Huffman coding to compress each of these fields in isolation using the methodology in SC^2 [5]. In particular, a Value-Frequency Table (VFT) [5] establishes the value-frequency statistics that are associated with the exponent and mantissa fields.

FP-H views a cache block as a row of consecutive, e.g. 8-byte, floating-point numbers, as depicted in the top chart of Figure 5. The alignment may differ, as depicted in the bottom chart of Figure 5, however, we only consider the first alignment scenario as relevant for the X86-64 architectures assumed in our evaluations. We next describe the compression and decompression processes.

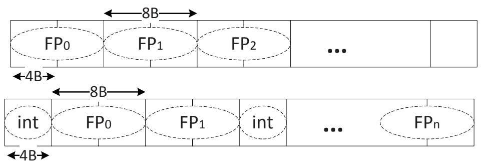

Figure 5: FP-alignment cases.

4.1 Compression

Compression is carried out in the following steps: I) Each of the floating-point numbers is broken down into four subfields: Sign (s), Exponent (e), Mantissa-High (mH) and Mantissa-Low (mL), which are grouped together as shown in the left part of Figure 6. II) Each subfield is compressed separately and in parallel by the respective compression engine. III) When data in all subfields are compressed (except for sign), they are concatenated together in a certain order to form the compressed block as shown at the bottom of Figure 6. Field grouping is not necessary, however, as various variable-length encodings are used this way of restructuring the data of the compressed block can dramatically accelerate decompression, as we show later. As we consider Huffman encoding for the compression of the mantissa subfields and for the exponent, the statistics for each subfield are monitored during the sampling phase using VFTs (e-VFT, mH-VFT and mL-VFT).

4.2 Decompression

A compressed block that is requested to be decom-

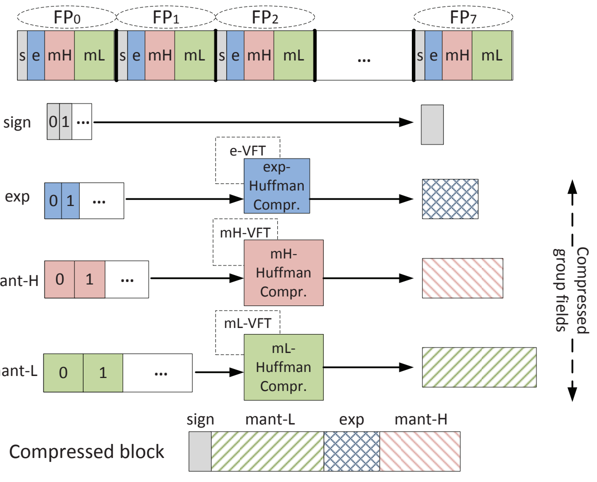

Figure 6: FP-H compression.

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 14:12:56 UTC from IEEE Xplore. Restrictions apply.

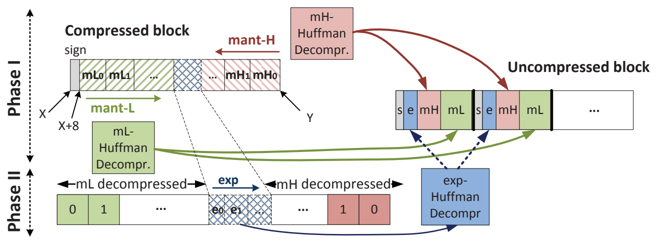

Figure 7: FP-H decompression. Two phases: (I) Simultaneous decompression for mL and mH, (II) decompression for exponent (e).

pressed is shown at the top of Figure 7. The boundaries of a compressed block are known (beginning: 'X'; end: 'Y') through the pointers (Section 3.4). We also know that the compressed mL group starts 8 bits after the group of uncompressed sign bits. However, there is no information about the exact offset for the compressed mH and exponent (e) fields, thus decompression for each of these fields must wait for the decompression of the previous field to complete first: decompress mL, then e and eventually mH. As Huffman decompression is inherently sequential, this could increase decompression latency significantly. As an alternative, we could keep metadata for the offsets of mH and e, however such area overheads may diminish the benefits of compression.

We propose instead a 2-phase decompression process: (I) Decompress mH and mL in parallel and then, (II) decompress the exponent. This is illustrated in Figure 7. We take advantage of the field grouping and save mH in reverse order in the end of the compressed block ('Y' pointer). This way we halve the decompression latency, as mL and mH are decompressed in parallel. Decompression for mH and mL will stop after each decompressor outputs 8 values. At that point, the boundaries of the compressed exponent field are also known, thus decompression can start immediately (second phase).

The FP-H decompressor makes use of the pipelined Huffman-based canonical decompressors as proposed in SC^2 [5]. The decompressed fields (mL, mH and e) are placed immediately in the respective bit positions of a fully decompressed block so that it is ready when the last exponent field is decompressed.

Finally, we note that the mL group is not always compressed well because of the irregularity of the mantissa's least significant bits. We find that this occurs in 45% of the cases, on average, across the simulated floating-point benchmarks. We can take advantage of this to further improve decompression latency in those cases, by requiring only phase I. A single bit (per block) keeps information about whether mL is encoded. The decompressor inspects this bit to decide whether mL decompression is needed, otherwise the exponent's decompression can start immediately and in parallel with mH decompression, as the group of exponents is located 8+32 x 8 = 264 bits further from the block's beginning.

4.3 FP-H-D: Faster FP Compression

The proposed FP-H compression method can be tailored in different ways depending on how we decide to compress the exponent. We have previously assumed that it is compressed using Huffman coding like the mantissa subfields. Although we manage to parallelize decompression for mL and mH, the inherently sequential decompression of the exponent in the second phase may still impact the cache access time.

We notice that each floating-point data set makes use of a few exponent values. Instead of densely encoding the exponent using Huffman encoding, we can use a lighter dictionary-based compression scheme that requires only at most two cycles in the second decompression phase. We name this variation *FP-H-D*. The dictionary is only updated during the sampling phase, while when Huffman encodings are generated for the mantissa subfields, we also freeze the dictionary for the exponent. The compressed exponent consists of a series of 8 index fields (4 bits per index), possibly followed by a variable number of uncompressed exponents, if they are not found in the dictionary. FP-H-D decompression is simple: each index is used to retrieve an exponent value from the dictionary. If the index is "0000", the uncompressed exponent is selected instead.

Table 1: Baseline configuration.

Cores	8 cores, X86-64, out-of-order, 3GHz Issue: 4, LQ: 64, SQ: 36, ROB: 168, RAS: 16
L1I/L1D L2	32 KB, 8-way, 2 cycles, 4 MSHR 256 KB, 8-way, 5 cycles, 32 MSHR
L3 (shared)	4 MB, 20 cycles, 16-way, 8 banks, 32 MSHR
Memory	200 cycles (66ns)

5. EXPERIMENTAL SETUP

5.1 Architecture Models

We evaluate HyComp and FP-H for both single-core and multi-core systems using the cycle-accurate GEM5 simulator [20] in syscall emulation using the classic memory system. The main baseline is a multi-core system with eight out-of-order cores, private L1/L2 caches per core and an L3 shared cache with eight banks as shown in Table 1. The block size is 64 bytes. The cache latencies are estimated using CACTI [21] and assuming a 3-GHz clock. The single-core system baseline uses a single-bank 1-MB LLC with a 10-cycle access time. This baseline helps us to systematically create multi-programmed workloads based on cache intensity and compressibility.

Compression methods: All compression methods listed in Table 2, including C-PACK+Z [4, 15], are integrated in the LLC. The tag and data arrays are decoupled; there are 4X more tags to realize a compression ratio of 4X. Extra tags and indirection adds two cycles according to CACTI simulations. Extra metadata are associated with each method as shown in Table 2. The use of the "Enc." bit(s) is described in Section 3.4 for HyComp and BDI. For SC^2 and FP-H, it determines the encoding version; for C-PACK+Z, it distinguishes between null blocks and CPACK-compressed ones. We note that with 48-bit addresses, the tag-area overhead of HyComp is only 3.8% over BDI.

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 14:12:56 UTC from IEEE Xplore. Restrictions apply.

Table 2: Tag size of a 4-MB, 16-way cache for baseline and compression schemes (#tags=4x).

	Tag	v,d,lru	index	c	Alg	Enc.	Total (KB)
Baseline	30	6	0	0	0	0	36 (288)
BDI	30	8	10	0	0	4	52 (1664)
SC^2	30	8	10	1	0	1	50 (1600)
FP-H	30	8	10	1	0	1	50 (1600)
HyComp	30	8	10	1	2	3	54 (1728)
CPACK+Z	30	8	10	1	0	1	50 (1600)

Table 3: (De)Compression latency per scheme. *In HyComp, it depends on the selected scheme.

	ZCA	SC^2	BDI	FP-H	HyComp	C-PACK+Z
Compr.	1	9	2	7	2+*	16
Decompr.	0	10	1	20/12	*	9

For each of the SC^2, FP-H, Brute-F and HyComp, the value-frequency statistics are collected by one VFT[2] for the whole cache, for the first 20-M committed instructions of a simulation phase. To make fair comparisons, we assume that the cache is not compressed by any scheme during this period. The same encoding is used for the rest of the simulation. Each LLC bank is associated with a compressor and decompressor for each compression scheme to avoid contention when there are simultaneous requests to the LLC from different cores. Moreover, the Huffman-based schemes make use of two decompressors to be able to decompress based on two different encodings [5]. Consequently, per LLC bank we have the following number of compressors/decompressors. SC^2: 1/4; FP-H: 3/6; FP-H-D: 2/4; Brute-F and HyComp: 4/10 Huffman-based (SC^2 and FP-H), 1/1 ZCA-based and 1/1 BDI-based. The HyComp heuristic is also modeled per bank in HyComp.

Baselines: We model and evaluate the following configurations: A 1-MB and 4-MB L3 physical (uncompressed) cache for the single-core and multi-core baselines used as references, respectively. We also compare the compressed cache schemes against 2X and 4X physically larger caches and analyze the sensitivity to the cache hit time; 8 MB (ideal/+8 cycles) and 16 MB (ideal/+8/+17 cycles) respectively, for multi-core; 2 MB (ideal/+4 cycles) and 4 MB (ideal/+8 cycles) respectively, for single-core systems.

5.2 Hardware Implementation

We have implemented and synthesized all the compression/decompression engines and the HyComp heuristic in VHDL using the Synopsys Design Compiler with 32-nm process technology and verified that they can be clocked at 3 GHz. The (de)compression latencies are summarized in Table 3. For C-PACK+Z, the latencies are derived from prior work [15].

SC^2: The decompressor contains combinatorial logic and a small 4-KB SRAM for the DeLUT [5] and forms a three-stage pipeline. Block compression and decompression take 9 and 10 cycles, respectively.

FP-H: A block is compressed in 7 cycles assuming dual-port compressors and one compressor per field (Exponent, Mantissa-High and Mantissa-Low). For the exponent the compressor/DeLUT has only 32 entries, thus can be implemented with logic. Hence, the exponent's decompressor has two pipeline stages. It takes 10 cycles to decompress 8 mL and mH fields (phase I), another 9 cycles to decompress the exponent (phase II) and 1 cycle to form the actual decompressed block, in total 20 cycles. If the mL field is not compressed, decompression can be completed in 11 cycles. On the other hand, in FP-H-D decompression takes 10 cycles in phase I and 2 cycles in phase II, in total 12 cycles.

[2]FP-H uses one VFT for each subfield

Table 4: Total area, leakage power (per bank) and dynamic energy (per access) for the baseline and compressed caches. (*HyComp's dynamic energy depends on the selected scheme)

L3 Cache	Area (mm^2) Total	St. Power (mW) per bank	Dyn. Energy (nJ) per Access	
Baseline	40.303	17.4	0.62	
2x Baseline	56.75	33.4	0.892	
4x Baseline	94.71	100	1.153	
Compressed	44.63	78.9	0.63	
Scheme	per bank	per bank	Compr.	Decompr.
ZCA	0.0007	0.001	0.0003	0
BDI	0.062	0.05	0.096	0.028
SC^2	0.138	6.76	0.21	0.168
FP-H	0.238	9.9	0.229	0.19
HyComp	0.452	16.62	*	*

BDI: Block compression takes 2 cycles and decompression one cycle.

ZCA: Block compression takes one cycle and decompression zero cycles.

HyComp: The heuristic takes two cycles; the first stage does the classification by doing the inspection of the block for all the types. The second stage takes as input the classification and by using simple bit-wise comparisons and priority encoding it makes a decision. A block's (de)compression latency depends on the selected scheme. Hence, the hit time of the compressed cache may vary. On a write/update, the cache hit time increases by "2+compr. lat". On a read/evict, it increases by the latency of decompressing the block by an amount corresponding to the compression algorithm used. This is modeled accurately in our evaluation.

Table 4 summarizes the area (in mm^2), the leakage power (per bank – in mW) and the dynamic energy (per access – in nJ) for the baseline cache, the compressed cache (accounting for the extra tag overheads), for the individual compression schemes we study and for HyComp. The cache area, power and energy are estimated using CACTI [21] for 32-nm technology. The LLC tag array is modeled with the ITRS-LOP cell and the data array with the ITRS-LSTP cell type. The reported area (for all 8 banks in caches) and leakage power (per bank) is a summary of both the tag and data arrays.

For the compression schemes the area and the leakage power are reported per bank but include both compressor and decompressor engines using the configuration discussed in Section 5.1. The dynamic energy is provided per access for the caches and per compression/decompression of a 64-byte block for each of the schemes. For example, FP-H requires 0.229nJ in total

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 14:12:56 UTC from IEEE Xplore. Restrictions apply.

Table 5: Multiprogrammed workloads.

Unaffected or Adversely affected		CR
mix1	cg-gromacs-milc-is	L
mix2	bwaves-lbm-soplex-leslie-gromacs-milc-namd-gamess	LMH
mix3	games-gromacs-milc-soplex-libq-gobmk-sjeng-perlb	LMH
Mixed effect		
mix4	cg-bt-mg-is-lu-sp	L
mix5	bt-bzip2-xalan-omnetpp-gamess-milc-lu-leslie3d	LH
mix6	mcf-omnetpp-gcc-xalan-cactus-astar-libq.-soplex	H
mix7	bzip2-astar-omnet-gcc-gamess-gobmk-perlben-zeus	MH
mix8	h264-hmmer-bzip2-is-lbm-perlbench-gromacs-namd	LMH
mix9	xalan-astar-namd-gromacs-gobmk-sjeng-gamess-lbm	LMH
Benefit		
mix10	astar-gcc-cactus-bzip2-zeusmp-h264-omnetpp-mcf	MH
mix11	hmmer-bzip2-h264ref-cactus-lu-xalan-sp-omnetpp	LMH
mix12	hmmer-sp-mg-cactus-astar-xalan-gcc-omnetpp	LMH
mix13	hmmer-bzip2-h264-cactus-astar-xalan-gcc-omnet.	MH
mix14	xalan-gcc-omnetpp-mcf-bzip2-astar-sp	LMH
mix15	xalan-gcc-omnetpp-mcf-cactus-astar-xalan-omnetpp	H

to compress all mL, mH and exponent fields in a 64-byte block and 0.19nJ for decompressing it. Hence, a compressed cache (0.63nJ per access) with FP-H consumes 38.5% and 32.2% more dynamic energy per access than an uncompressed cache for compression and decompression respectively, but 3.5% and 8.1% less energy/access than a 2X physically larger cache, and 25.4% and 28.8% less energy/access than a 4X larger one.

In HyComp, the dynamic energy for (de)compression of a block depends on the selected scheme, while the heuristic adds 5pJ more dynamic energy to each compression. The area overhead for the compression/decompression logic and the heuristics in all the banks is 8% of a 4-MB compressed cache, while the leakage power because of compression/decompression is only 17% of the total leakage power of this compressed cache.

5.3 Metrics

To evaluate the efficiency of the compression methods, we use Compression Ratio= $\frac{Uncompressed\ set\ size}{Compressed\ set\ size}$ for each cache set and then calculate the average across all the sets. The accuracy of HyComp's heuristic is measured by comparing the decision made by HyComp to Brute-F, which always does the right decision. Performance is evaluated using Misses per Kilo Instruction (MPKI) in the L3 cache and Speedup of execution time for the single-core system and Weighted speedup [22] for the multi-core system, using as reference the baseline.

5.4 Workloads

We use 28 benchmarks: 22 SPEC2006 [23] (12 integer and 10 floating-point ones) and 6 NAS benchmarks (cg, mg, bt, lu, sp and is) compiled with gcc (optimization flag -O2) or Fortran. SPEC2006 applications are run using the reference input while the NAS benchmarks run with the B problem class. We simulate our system for representative phases that were defined using pinpoints [24]: 10-15 pinpoints per application of duration of 250M committed instructions each.

Classification of workloads: Benchmarks are classified based on the impact of the increased LLC capacity on their performance, yielding three groups: 1) Applications that *benefit* from more cache capacity, 2) Applications that remain *unaffected* and 3) applications that are *adversely* affected because of longer access time due to larger cache. Based on the classification, we create 15 multiprogrammed workload mixes (mix1 to mix15) to use in the multi-core simulations, as Table 5 shows. The mixes are created also based on the compression ratio (L,M,H[3]) of individual applications, based on our findings in Section 6.2.1, similarly to prior work [3, 5]. Most mixes contain 8 workloads except for mix1, mix4 and mix14. Mix1 consists of compute-intensive workloads with low compressibility; mix4 has only NAS benchmarks, while mix14 involves the most cache-intensive and most highly compressible applications.

Before a detailed multi-core simulation is executed, we fast-forward the execution 5-billion committed instructions and warm up the caches for another 500-M committed instructions, similarly to prior work [25]. Detailed simulation is then done until all applications have committed 250-M instructions. When a benchmark completes 250-M instructions, we collect the statistics and keep it running to stress the shared resources until all of them complete 250-M instructions.

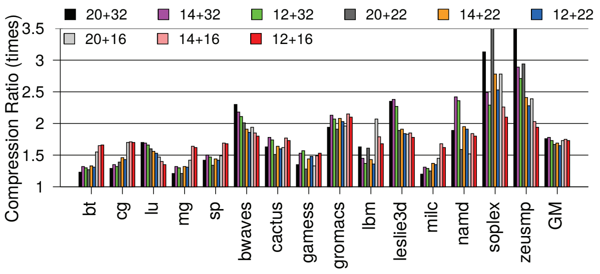

Figure 8: Compressibility of the mantissa.

6. EVALUATION

6.1 Value Locality in the Mantissa

Previous studies find that the exponent of floating-point values exhibits high value locality [9, 10, 11] and typically exhibits a compression ratio of 4X, whereas the mantissa exhibits low value locality. However, the exponent comprises only 17% of a floating-point number and offers overall a low compression ratio. If value locality in the mantissa can be uncovered, it would lead to a substantially higher overall compression ratio. To verify this, we study the overall compression ratio of FP numbers by dividing the mantissa into two subfields (Mantissa-High and Mantissa-Low) and compress them individually so as to establish the size of each of them that maximizes the compression ratio.

Figure 8 shows the compression ratio for a 1-MB cache (null blocks are removed) for the mantissa assuming its 20 most-significant bits for Mantissa-High and its 32 least-significant bits for Mantissa-Low by varying their widths. The compression ratio is derived by dividing the number of mantissa values using 52 bits by "$\#UniqueVal \times H + \#TotalVal \times (20 - H) + \#UniqueVal \times L + \#TotalVal \times (32 - L)$", where H and

[3]L: $CR < 1.5X$; M: $CR \leq 2X$; H: $CR > 2X$

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 14:12:56 UTC from IEEE Xplore. Restrictions apply.

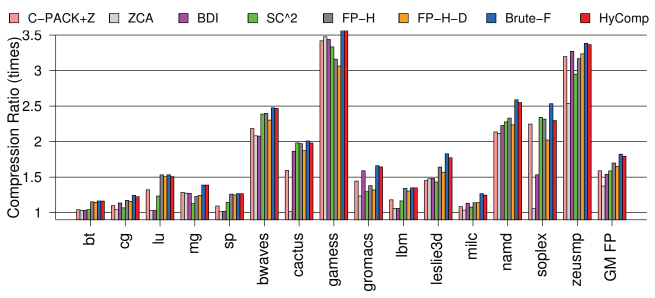

Figure 9: Compressibility for FP applications.

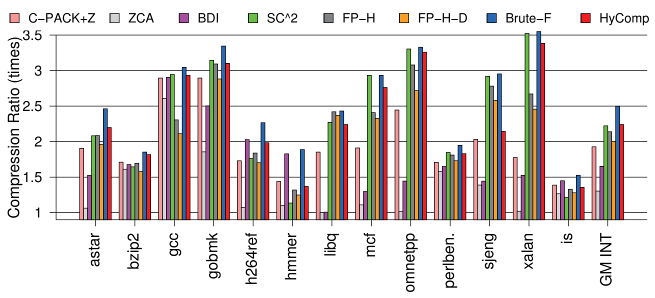

Figure 10: Compressibility for INT applications.

L are the number of bits considered to form Mantissa-High and Mantissa-Low values, respectively. For example, "14+22" shows the compressibility of the mantissa by only considering the first 14 bits of Mantissa-High and the first 22 bits of Mantissa-Low to form unique values, respectively, while 6 and 10 bits for all values are left uncompressed, respectively. The geometric mean (GM) shows that by taking into account the whole mantissa sub-fields, but extracting value locality in isolation, compression is close to 1.8X. However, for NAS benchmarks (except for lu), best results are achieved by disregarding the 16 LSB of Mantissa-Low.

6.2 Results for Single-Core Systems

6.2.1 Compressibility Results

Figures 9 and 10 show the compression ratio across the floating-point (FP) and the integer applications, respectively. Brute-force (Brute-F) shows the upper bound on compressibility of hybrid compression. The maximum compression ratio is 4X, as we restrict the number of tags to 4x per cache set. Cache compressibility is calculated using a physical 1-MB L3 cache as a baseline.

Figures 9 and 10 confirm that there is no single compression method that always outperforms others. For example, FP-H (and its variation FP-H-D) improves the compressibility for some FP applications: lu, sp, bt, cg, lbm and leslie3d, when compared to prior work. ZCA is more efficient when null blocks are common, e.g., gamess and namd. On the other hand, SC^2 is more capable for common 32-bit values, e.g., in integer benchmarks mcf, xalan and omnetpp. Interestingly, FP-H compresses integer workloads quite well too. If value locality is exhibited in 32-bit portions this can be captured at finer granularities exploited by FP-H, but not to the extent that SC^2 does. In few FP applications, e.g., cactus and soplex, SC^2 and FP-H compress equally well. Moreover, in gromacs, is (i.e., "integer sort" of NAS), hmmer and h264ref, BDI offers best compression followed by C-PACK+Z, which supports partial dictionary matches. The first two applications use a lot of pointers (array-index values), while the other two manipulate close or repeating 32-bit integer values.

Using an ideal hybrid scheme – Brute-F – we verify that each application contains different data types which means that a hybrid scheme not only has a potential to perform more robustly across different applications, but also can outperform a single scheme for a single application. More importantly, when HyComp is used, compression is in most cases close to Brute-F and better than individual schemes, on average.

The few cases where HyComp does not outperform individual compression methods are: (a) mcf, sjeng and xalan (best: SC^2); (b) hmmer and is (best: BDI); and (c) libq (best: FP-H). This is attributed to the lower accuracy of the heuristic in predicting the right scheme. For each application, the two leftmost bars in Figure 11 show the accuracy of HyComp and a heuristic of prior work [7] using Brute-F as a reference (100 % accuracy). The two rightmost stacked bars show the distribution of the selected schemes for HyComp and Brute-F. For example, in mcf the deviation (about 10%) is attributed to blocks predicted as pointer but were left uncompressed by BDI due to multiple ranges, while Brute-F selects SC^2 as is revealed by comparing the two stacked bars.

HyComp exhibits 80% accuracy in selecting the best compression method, on average, while a prior work [7] is limited to 60%, on average. The worst accuracy for HyComp is noticed for hmmer and soplex. In hmmer, there are many 32-bit values with 0xFF in their MSB, likely negative integer. Based on the stacked bars, the heuristic predicts them as floating-point (FP-H), but Brute-F uses BDI. In soplex, the value 1 and other small integers represented as FP[4] occur in the cache by 40%, on average. The heuristic chooses FP-H but Brute-F selects SC^2 as this way they are compressed more densely.

In summary, we have shown that hybrid compression can combine the benefits of individual compression methods to realize a compression solution that offers higher compressibility robustly. The results show that HyComp, on average, outperforms the individual schemes. Furthermore, an improved heuristic would result in even larger improvement.

6.2.2 Performance Results

Figure 12 shows the speedup for the single-core workloads for the different individual compression methods (including C-Pack+Z [4, 15]) and the hybrid schemes: Brute-F and HyComp[5] (Hycomp-free has no (de)compression latency – upper bound). The speedup is calculated using as reference a physical 1-MB cache. We also compare against 2X and 4X physically larger caches (the

[4]This creates regularity in the mantissa and exponent.

[5]As we observe better compression and speedup with FP-H than FP-H-D, we only explore HyComp with FP-H.

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 14:12:56 UTC from IEEE Xplore. Restrictions apply.

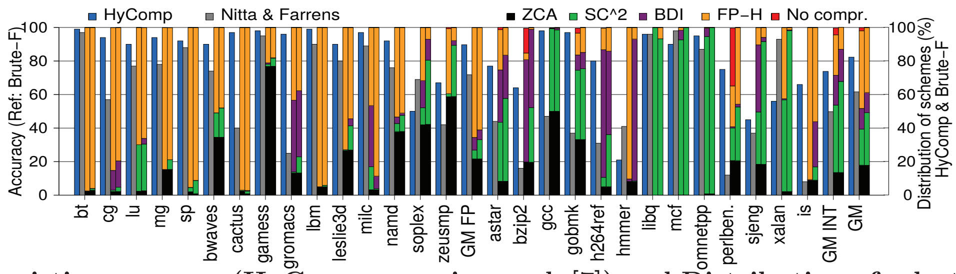

Figure 11: Heuristics accuracy (HyComp vs. prior work [7]) and Distribution of selected schemes for HyComp vs. Brute-F (stacked bars)

leftmost bars) with and without considering the extra latency imposed. The latter cases are denoted as "ideal".

Figure 12 shows that hybrid compression does better than individual methods, as Brute-F offers higher performance robustly for all applications, on average. HyComp, nonetheless, follows this trend by offering better performance for integer workloads as compared to individual methods, on average. Specifically, in many cases, HyComp picks the method that gives best improvement to the system (e.g., mcf, omnetpp, bzip2 and astar). When considering 2X and 4X larger caches, for all floating-point and few integer applications (bzip2, h264ref and hmmer), HyComp and other methods never exceed the performance of a 2X larger cache, as compression ratio rarely exceeds 2X. For the rest though, HyComp performs close to a 4X larger cache.

It is important to note that HyComp neither improves nor penalizes the performance of most floating-point workloads[6] as well as a few integer-workloads[7] when compared to individual methods. This is because most of them do not benefit from larger caches. Among the floating-point applications though, we notice improvements for lu and sp, thanks to the proposed FP-H compression scheme. The substantial speedup for sp is because MPKI is improved from 18 to 14 with FP-H and HyComp. The loss in cg is because of the increased cache hit time without any benefit from larger capacity as is revealed by the baselines (ideal vs. normal).

In one case (gcc), the individual methods BDI performs better. While Brute-F and HyComp select the scheme that yields the highest compressibility (i.e., ZCA and SC^2 based on Figure 11), this does not yield the best speedup, as SC^2 imposes longer decompression latency than BDI mitigating the negligible gains in compression ratio (see Figure 10).

In summary, while individual compression methods can achieve better performance in some cases, we have shown that hybrid compression offers better performance on average. Specifically, HyComp outperforms individual methods and achieves better performance for all workloads independently of the data types they use.

6.3 Results for Multi-Core Systems

For the multi-core simulations, we run the multiprogram mixes of Table 5 on a multi-core system with eight cores. Figure 13 shows the speedup for the different mixes for HyComp and individual schemes as well as 2X and 4X physically larger caches (both ideal and normal, as in Section 6.2.2). The results are normalized to the baseline system with a 4-MB physical cache.

Overall, we observe that HyComp outperforms individual compression methods and performs nearly as well as Brute-F. When considering physically larger caches (the five leftmost bars), we observe that HyComp has better speedup than a 2X baseline except for mixes with low sensitivity and a few medium ones. In addition, while a 4X ideal cache outperforms HyComp and individual schemes, it performs similar to Hycomp-free. This reveals that hybrid compression is so effective that it can reach the performance of 4X larger cache when the latency overheads are disregarded. However, when the cache hit time grows due to larger capacity by 8 and 17 cycles, HyComp performs similar or better, respectively. In the latter case, the reason is that the access time of the cache becomes longer than the average decompression latency of HyComp.

Let us now investigate results for individual workload mixes. In the first three mixes, we see that the performance remains unaffected. This is expected for mix2 and 3, but in mix1 we observe that HyComp may be affected a bit negatively due to cg. In mix4, FPH and HyComp neither show benefit nor adverse impact as the losses in cg are compensated by the gains in sp, while for the rest of the schemes, performance remains unaffected as they don't compress. Although mix5 contains xalan and omnetpp, that benefit from compression, their benefit compensates for the loss in leslie3d and bt.

As we go to the right, more applications involved in the mix are compressed better. The higher compressibility for HyComp is accounted for by the fact that it picks the best algorithm for gcc and omnetpp, while BDI and SC^2 compensate their gains with losses from each other. In mix8, the deviation between HyComp and Brute-F is attributed to the inaccuracy to pick BDI for hmmer, as in mix9 which has almost the same configuration as mix8, HyComp performs similarly to Brute-F. The advantage of HyComp is clearly articulated in the third batch of workloads mixes that mainly consists of memory-intensive workloads.

As for cache energy consumption, a 4-MB compressed cache with HyComp exhibits 77% higher dynamic energy than a 4-MB uncompressed cache, but only 5% more than a 2X physically larger and 23% less than a 4X larger cache, across all workload mixes. However, due to compression, HyComp reduces the number of

[6]bt, bwaves, gamess, gromacs, lbm, milc and namd.
[7]libquantum, gobmk and sjeng.

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 14:12:56 UTC from IEEE Xplore. Restrictions apply.

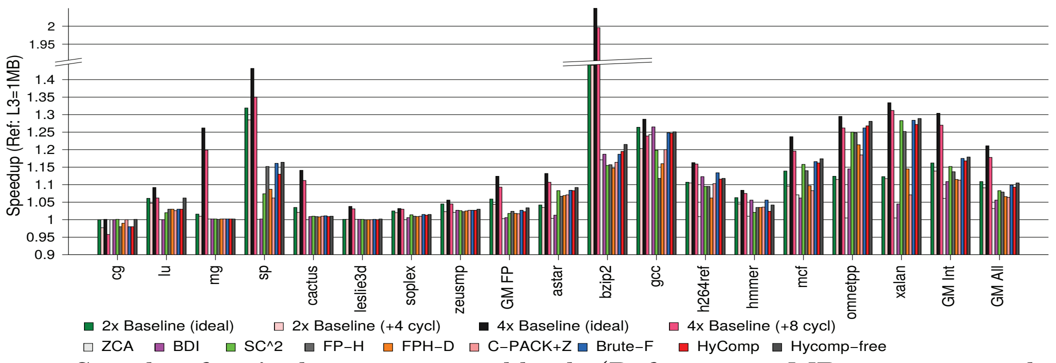

Figure 12: Speedup for single program workloads (Reference: 1-MB uncompressed cache).

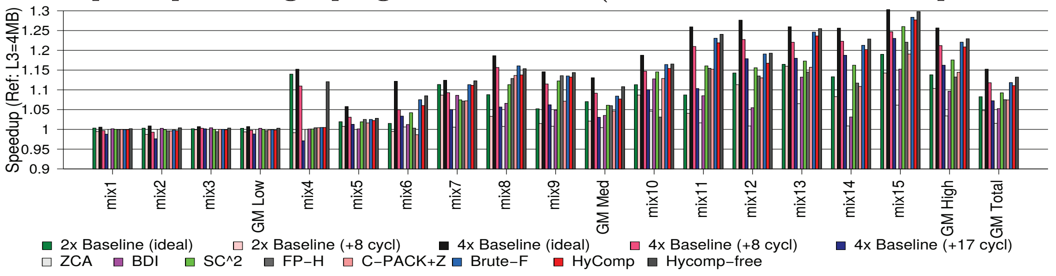

Figure 13: Weighted speedup for multiprogram workloads (Reference: 4-MB uncompressed cache).

off-chip accesses. By assuming 20nJ to bring a 64-byte block from memory (i.e., more conservative than prior work [5]), HyComp is 39% more energy efficient than the 4-MB baseline, on average, and, 16% and 7.5% than the 2X and 4X larger caches, respectively. Compared to Brute-F, although HyComp has slightly lower compression ratio than Brute-F, HyComp is 5% more energy efficient, on average, as Brute-F requires 22% more dynamic energy to try all compression schemes. In conclusion, HyComp performs closely to Brute-F and better than prior work.

7. RELATED WORK

Concerning hybrid compression and data-type prediction, Kant and Iyer [8] and Nitta and Farrens [7] contribute with a data-type prediction heuristic that, similarly to ours, can distinguish between integers, pointers and floating-point numbers, but is used for the purpose of optimizing link compression and not to select compression methods. The heuristic in those works are similar to each other but different to HyComp. The former [8] makes a decision in a per-word basis. This is unsuitable for cache compression as a block compressed with various methods requires metadata imposing extra area and decompression latency. Sardashti and Wood [15] study the combination of two specific compression schemes: Null-block combined and C-PACK (i.e., C-PACK+Z). They do this for evaluation purposes rather than exploring its design space. In contrast, this work contributes with, for the first time, how to combine the benefits of individual compression methods.

Regarding floating-point compression, Citron [9] finds that only the most significant byte of a double-precision floating-point number has low entropy and can be compressed effectively. Recently, Townsend and Zambreno [11] verify this by comparing the most significant byte between adjacent floating-point values. Unfortunately, the exponent occupies a small fraction of a floating-point number. We show, for the first time, that there is ample value locality also in the most significant portion of the mantissa and how it can be practically exploited. Several proposed floating-point compression methods rely on prediction. Isenburg et al. [26] use exponents to select among different arithmetic contexts to predict the mantissa. Lindstrom and Isenburg [27] use prediction and range encoding to exploit the fact that IEEE-754 floating-point numbers in 3D grids are spatially close. They note that double-precision FP values cannot be predicted. FPC [14], as we evaluate in this paper, compresses double-precision floating-point numbers by relying on the history in order to predict future values. This does not work well in context of randomly accessed memory structures (e.g., caches). Finally, Sathish et al. [28] explore lossy compression in the low order bits of the mantissa in GPGPU workloads and notice a small impact on accuracy. FP-H, on the other hand is a lossless compression method that does not rely on access order, thus is well adapted for caches.

8. CONCLUSIONS

This paper first contributes with hybrid compression in which one, out of several data-type specific compression methods, is selected based on dynamic prediction of the data type with an accuracy as high as 80%. This yields a substantially higher compression ratio compared to prior work. In addition, HyComp only adds marginally to the compression latency (2 clock cycles) and does not affect the decompression latency. As a result, we show that HyComp can offer significant speedup for a number of multiprogrammed workloads.

The second contribution is the finding that there is ample value locality in the high-order bits of the man-

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 14:12:56 UTC from IEEE Xplore. Restrictions apply.

tissa. Based on that, the paper proposes a new hardware-based floating-point method (FP-H) that compresses/decompresses the exponent and the mantissa in parallel, using Huffman encodings. This paper shows that HyComp together with FP-H can compress prevailing data types such as integers, pointers and floating-point numbers effectively and robustly.

9. ACKNOWLEDGEMENTS

We would like to thank Chloe Alverti and the anonymous reviewers for their insightful feedback. We acknowledge Somayeh Sardashti for providing the code for C-PACK+Z, and Jonas Andersson and Niklas Doverbo for providing the area / energy numbers for BDI. This work was funded by the European Union under the project EUROSERVER (no: 610456), and in part by an advanced grant from the European Research Council (ERC) through the MECCA project (no: 340328) and by the Swedish Foundation for Strategic Research (SSF) under the project SCHEME (RIT10-0033). The simulations ran on the resources provided by the Swedish National Infrastructure for Computing (SNIC) at C3SE.

10. REFERENCES

[1] J. Dusser, T. Piquet, and A. Seznec, "Zero-content augmented caches," in *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pp. 46–55, ACM, 2009.

[2] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pp. 212–, IEEE Computer Society, 2004.

[3] 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 *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pp. 377–388, ACM, 2012.

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

[5] A. Arelakis and P. Stenstrom, "SC^2: A statistical compression cache scheme," in *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ISCA '14, pp. 145–156, IEEE Press, 2014.

[6] E. Hallnor and S. Reinhardt, "A unified compressed memory hierarchy," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pp. 201–212, Feb 2005.

[7] C. Nitta and M. Farrens, "Techniques for increasing effective data bandwidth," in *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pp. 514–519, Oct 2008.

[8] K. Kant and R. Iyer, "Compressibility characteristics of address/data transfers in commercial workloads," 2002.

[9] D. Citron, "Exploiting low entropy to reduce wire delay," *IEEE Comput. Archit. Lett.*, vol. 3, pp. 1–1, Jan. 2004.

[10] L. Gomez and F. Cappello, "Improving floating point compression through binary masks," in *Big Data, 2013 IEEE International Conference on*, pp. 326–331, Oct 2013.

[11] K. Townsend and J. Zambreno, "A multi-phase approach to floating-point compression," in *Proceedings of the IEEE International Conference on Electro/Information Technology (EIT)*, May 2015.

[12] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, pp. 1:1–1:25, Dec. 2011.

[13] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS parallel benchmarks – summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, (New York, NY, USA), pp. 158–165, ACM, 1991.

[14] M. Burtscher and P. Ratanaworabhan, "FPC: A high-speed compressor for double-precision floating-point data," *Computers, IEEE Transactions on*, vol. 58, pp. 18–31, 2009.

[15] S. Sardashti and D. A. Wood, "Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pp. 62–73, ACM, 2013.

[16] S. Sardashti, A. Seznec, and D. A. Wood, "Skewed compressed caches," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pp. 331–342, 2014.

[17] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the Institute of Radio Engineers*, vol. 40, pp. 1098–1101, Sept 1952.

[18] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pp. 74–85, IEEE Computer Society, 2005.

[19] Y. Tian, S. M. Khan, D. A. Jiménez, and G. H. Loh, "Last-level cache deduplication," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, (New York, NY, USA), pp. 53–62, ACM, 2014.

[20] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[21] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," technical report hpl-2009-85, HP Laboratories, 2009.

[22] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, ASPLOS IX, pp. 234–244, ACM, 2000.

[23] C. D. Spradling, "SPEC CPU2006 Benchmark Tools," *SIGARCH Computer Architecture News*, vol. 35, 2007.

[24] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large intel itanium programs with dynamic instrumentation," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pp. 81–92, 2004.

[25] D. Sanchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," in *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pp. 57–68, ACM, 2011.

[26] M. Isenburg, P. Lindstrom, and J. Snoeyink, "Lossless compression of predicted floating-point geometry," *Comput. Aided Des.*, vol. 37, pp. 869–877, July 2005.

[27] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 12, pp. 1245–1250, Sept 2006.

[28] V. Sathish, M. J. Schulte, and N. S. Kim, "Lossless and lossy memory i/o link compression for improving performance of GPGPU workloads," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pp. 325–334, ACM, 2012.

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 14:12:56 UTC from IEEE Xplore. Restrictions apply.