A Case for Partial Co-Allocation Constraints in Compressed Caches

Daniel Rodrigues Carvalho, André Seznec

To cite this version:

Daniel Rodrigues Carvalho, André Seznec. A Case for Partial Co-Allocation Constraints in Compressed Caches. SAMOS XXI 2021 - International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, Jul 2021, Samos, Greece. pp.1-13. hal-03284824

HAL Id: hal-03284824

https://hal.inria.fr/hal-03284824

Submitted on 12 Jul 2021

A Case for Partial Co-Allocation Constraints in Compressed Caches

Daniel Rodrigues Carvalho[1][0000−0001−7919−9347] and
André Seznec[2][0000−0002−3058−6503]

Univ Rennes, Inria, CNRS, IRISA, Rennes, France
[1]odanrc@yahoo.com.br
[2]andre.seznec@inria.fr

Abstract. Compressed cache layouts require adding the block's size information to the metadata array. This field can be either constrained — in which case compressed blocks must fit in predetermined sizes; thus, it reduces co-allocation opportunities but has easier management — or unconstrained — in which case compressed blocks can compress to any size; thus, it increases co-allocation opportunities, at the cost of more metadata and latency overheads. This paper introduces the concept of partial constraint, which explores multiple layers of constraint to reduce the overheads of unconstrained sizes, while still allowing a high co-allocation flexibility. Finally, Pairwise Space Sharing (PSS) is proposed, which leverages a special case of a partially constrained system. PSS can be applied orthogonally to compaction methods at no extra latency penalty to increase the cost-effectiveness of their metadata overhead. This concept is compression-algorithm independent, and results in an increase of the effective compression ratios achieved while making the most of the metadata bits. When normalized against compressed systems not using PSS, a compressed system extended with PSS further enhances the average cache capacity of nearly every workload.

Keywords: Cache · Hardware Compression · Cache Organization.

1 Introduction

Cache compressors process data in uncompressed format to generate compressed output. Typically, compressors focus on reaching good compression factors or fast decompression latencies to improve system performance or cache capacity [17]. However, compression by itself is not enough to achieve these goals; a *compaction scheme* (or *cache organization*, or *compactor*) must be used to determine what to do with the compressed data. That is, compaction schemes expand the capabilities of conventional tag-data mapping methods to account for compressed blocks and their ability to share data entries.

Some compaction techniques limit compression to fixed sizes (*e.g.*, 25% and 50% of the line size), adding padding to lines smaller than these sizes [18, 19]. These *constrained* methods have low metadata overhead, but limit co-allocation by removing opportunities. Moreover, while cache compressors may be successful in some workload regions, there is still plenty of data that fails to attain favorable compressed sizes for compaction; the average compressed size in SPEC 2017 for multiple state-of-the-art compressors [1,3,4,6,11,12,15], is still far above 50% of the uncompressed size (Figure 1), making it hard to effectively co-allocate blocks with such limitations.

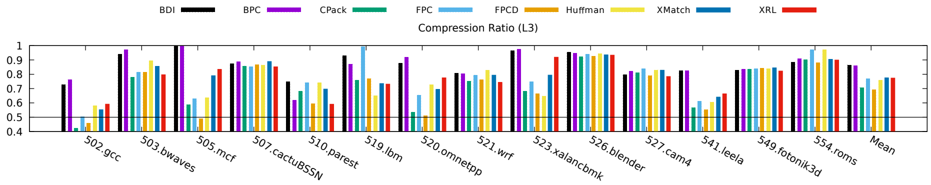


Fig. 1. Average compression ratio of SPEC 2017 workloads for multiple state-of-the-art cache compression methods applied to the Last-Level Cache (L3). Lower is better.

Other proposals remove these limits, allowing blocks to be compressed to any size [2,6] — a concept we will refer to as *unconstrained* methods. Although these methods allow compression to reach its full potential, they significantly increase metadata overhead due to the number of bits needed to represent the compressed size. Besides, locating lines becomes non-trivial: they can be found anywhere in the data entry. This results in a few more cycles being added to the access path.

We have come up with **Pairwise Space Sharing** (**PSS**), a technique that achieves the best trade-off between limiting the number of possible sizes and having an unconstrained representation. PSS introduces the notion of a partially-constrained representation: blocks are stored in groups of two — *block pairs* — and although each pair must fit in a fixed-size entry, the blocks within a pair have less restrictions. As a result, **PSS keeps line location trivial, and requires far less metadata bits than conventional unconstrained methods, while still making the most out of co-allocation opportunities**. Moreover, Pairwise Space Sharing can be applied in conjunction with most state-of-the-art cache compaction proposals.

This paper makes the following contributions:

- We show a particular case of size constraints that significantly reduce metadata and latency overheads of existing methods.
- We demonstrate that having a fully unconstrained representation is sub-optimal when the compression design is focused on neighbor-block co-allocation.
- We group these benefits to propose Pairwise Space Sharing, an expansion to compaction layouts which allows the benefits of unconstrained compaction with minimal tag overhead and no extra latency.

The following terms will be used throughout this paper: **compression ratio** is the ratio between the compressed size and the cache line size [16]; and **compaction ratio** — also referred to in the literature as *effective cache capacity* —

is the number of valid blocks in a data entry. The former measures how *efficient* a compressor is, while the latter exposes the *efficacy* of the compressed system (compression + compaction).

2 Size Constraints of Compressed Blocks

After a block is compressed, a compaction method is used to determine if it can be co-allocated with other block(s). These techniques may use different approaches to decide how to co-allocate, but they must always rely on one piece of information: the block size. Co-allocated blocks must fit in their designated space, which means that each block's size must always be retrievable; thus, a compressed block's size is an inherent metadata overhead. Sizes are typically represented at a byte granularity to slightly reduce this overhead [2, 6].

The size field can be either **unconstrained** — *i.e.*, all sizes are possible — or **constrained** — *i.e.*, compressed blocks are padded to fit in predetermined sizes. While unconstrained sizes are theoretically ideal to make the most out of compression, they come at a high cost: large metadata overhead. Furthermore, its placement process is fairly complex due to varying available sizes and higher number of location possibilities, which may require a few extra cycles to process.

Constrained sizes use larger granularities to ease these drawbacks — *e.g.*, at half-line granularities a compressed block can either be compressed to half or be left uncompressed, which would require a single-bit size field, and generate only two possible block locations — but add a penalty to the compression efficiency: data entries routinely end up wasting many bits with padding [18, 19].

2.1 Partial Constraint

We hereby define a third possibility: **partially constrained** sizes. *A partially constrained entry is split into multiple constrained segments, and each of those segments uses an unconstrained layer.* For example, a 512-bit data entry can be divided into four 128-bit segments. Each segment can co-allocate blocks without constraints, as long as they fit in its 16-byte space, as depicted in Figure 2.

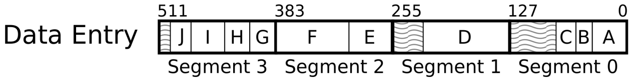


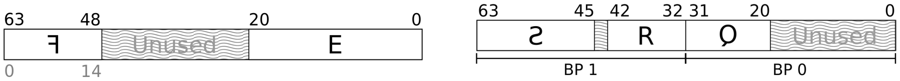
Fig. 2. Example of a data entry split into four constrained segments. Each segment co-allocates blocks in an unconstrained fashion: blocks can be compressed to any size, as long as their sum fits in their segment's 128 bits.

One possible goal of smaller constrained entries is to allow restricting placement. If a rule is applied so that, for example, a given block B can only be assigned to S specific segments, then $\frac{P-S}{P}$ of the P placement locations are removed from the possibilities. This reduces the number of size bits by $\lfloor \log_2(P - S) \rfloor$. Nonetheless, this restriction is not enough to satisfy latency requirements, because B can still be stored anywhere within its allowed segments, which may still be a large gamma of placement possibilities.

2.2 A Case for Latency-Efficient Partial Constraint

There are, however, two special cases of partial constraint that deserve distinct attention: when the number of blocks allowed per segment are, respectively, 1 and 2. When only one block can be allocated per segment, there are two possibilities: either it is an uncompressed cache (the segment size matches the cache line's size); or it is a general constrained method — the segment size is smaller than the cache line's, and compressed blocks must fit in fixed-sized entries.

The other case, when there are up to two blocks per segment, has a peculiarity that can be exploited to greatly simplify locating blocks. Within a constrained segment, no matter its size, there are two invariable locations: its leftmost bit, and its rightmost bit (*i.e.*, the extremities). These can be used as markers that define the beginning of a sub-block, with one of the sub-blocks being stored in reverse order (the MSB becomes the LSB and vice-versa) (Figure 3a). *Since these locations are statically defined, there is no latency overhead to locate blocks within a segment.* We will refer to segments that contain up to two blocks as a **block pair** (**BP**).



(a) Block placement in an entry containing a single BP. E is stored conventionally, and F is stored with its bits reversed.

(b) A data entry with two segments supports up to four sub-blocks — there are two BPs. R and S are paired up in a BP 1, and Q is in BP 0. Q's companion is not present. Q and S are stored reversed.

Fig. 3. Overview of the sub-block placement in BPs. Each sub-block is stored relative to an extremity of its BP.

Another advantage of having fixed extreme locations is that, since the bits in between the sub-blocks are unused, *data contractions and expansions that still fit in the pair do not need recompaction.*

3 Pairwise Space Sharing

We herewith introduce **Pairwise Space Sharing** (**PSS**): *a partially constrained co-allocation technique that uses block pairs (BPs).* Contrary to previous approaches, *Pairwise Space Sharing stores metadata implicitly, and reduces the* *likelihood of data expansions* — *i.e.*, it reduces the metadata overhead, yet yields better results. PSS is independent of the space available; thus, it can be applied to compressed cache layouts that allow more than two blocks per data entry.

For instance, in YACC [19] a superblock's compression factor (CF) defines the minimum size a sub-block must attain to be compressed: a quarter ($CF = 4$), and half of the data entry ($CF = 2$). If, for example, PSS is applied on top of YACC, this limitation is lessened, and the ***pair's size*** *must fit in a half or a whole data entry instead*, respectively. No further modifications are needed, and sub-blocks are paired like in YACC: when $CF = 2$ there is only one BP, and any of its four sub-blocks can be paired in it (Figure 3a); when $CF = 4$, there are at most two pairs, and each sub-block's position in the data entry is fixed — sub-block 0 can only be paired with 1, and sub-block 2 with 3. Figure 3b shows an example of data entry containing more than one BP.

It is important to notice, however, that PSS is not limited to a YACC-like design, so it could handle placement differently (*e.g.*, by adding a position field to the metadata when the CF is 4 too to allow sub-blocks to be placed in any of the available extremities — see Section 3.4 for more information on that). In short, PSS decides *where* and *how*, not *which* blocks are allocated in a data entry; thus, it can be applied to non-superblock-based layouts too.

3.1 Decreasing the Number of Unsuccessful Co-Allocations

A BP's size is fixed, but dependent on the CF (see Equation 1). For example, given 64B cache lines, a superblock with $CF = 2$, has one BP; so, the size fields of its two blocks would naively require $2 \cdot 6$ bits; when $CF = 4$, two BPs can reside in the superblock, so $4 \cdot 5$ bits would be needed for the size fields, per tag entry. This naive approach assumes that any size is valid; however, the probability distribution of compressed sizes follows a non-uniform cumulative distribution function: barely compressing a block is significantly more frequent than compressing it to a tiny size (as seen in Figure 1).

$$BPSize_{CF} = 2 \cdot \frac{cacheLineSize}{CF} \tag{1}$$

Consequently, *a large block will likely not co-allocate, and impose an unnecessary decompression latency fee on hits.* This observation is especially true for superblock-based compaction methods, since neighbor blocks tend to have similar compressibility [14], so a large sub-block will probably have a comparably sized counterpart. *Hence, one can limit the range of possible sizes within a segment to increase the likelihood of co-allocating blocks.*

Sizes are stored as a number relative to $minSize_{CF}$. By limiting the range of valid compressed sizes ($[minSize_{CF}$:$maxSize_{CF}]$, with values respecting Equation 2), *not only does the likelihood of having non-co-allocated blocks reduce, but also the size field's width is also decreased* (Equation 3 if $2 \cdot maxSize_{CF} \neq BPSize_{CF}$; 0 otherwise). This means that sizes are stored as numbers relative to $minSize_{CF}$. As a consequence, blocks whose size is smaller than $minSize_{CF}$ are rounded up to $minSize_{CF}$.

$$minSize_{CF} = BPSize_{CF} - maxSize_{CF} \tag{2}$$

$$sizeBits_{CF} = \log_2 \left(2 \cdot maxSize_{CF} - BPSize_{CF}\right) \tag{3}$$

In the previous example, if the **size-field range** is set so that $maxSize_{CF} = 62.5\% \cdot BPSize_{CF}$ of the uncompressed line ($minSize_2 = 24B$, $maxSize_2 = 40B$, $minSize_4 = 12B$, $maxSize_4 = 20B$), the width of a size entry is reduced to $sizeBits_2 = 4$, and $sizeBits_4 = 3$. Therefore, an absolute size of 30B would be stored as a relative size of 6B ($30B - minSize_2 = 6B = \mathit{0110}_2$), and an absolute size of 10B would be stored as 0B.

3.2 Halving the Number of Size Fields

Since the segment's size and location, and the blocks' location are always known, one can further reduce the size-related metadata overhead: *only one of the sub-blocks' sizes needs to be stored in the tags, in the pair's respective size field entry*, and the other (e.g., the non-reversed sub-block's) can be implicitly estimated as its complement. If only the non-reversed block is present in the pair, the stored size represents the available space for the reversed sub-block.

This optimization has the drawback that the reversed block must be decompressed and re-compressed to retrieve the real available size in the BP whenever its companion suffers a data expansion (*i.e.*, a write larger than its current size occurs). Figure 4 presents how the size field is interpreted under different CFs, for both non-optimized and optimized configurations. Nonetheless, this event is rare, not on the critical path, and the re-compression step can be removed by adding a delimiter code to the end of the companion's compressed data.

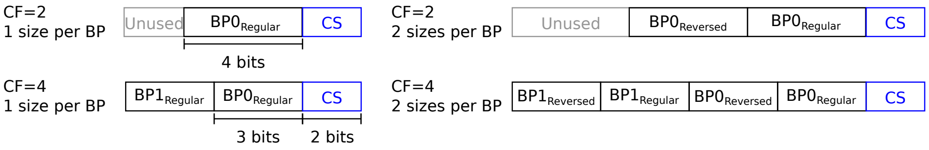


Fig. 4. A comparison of size-related metadata used at different compression factors for different configurations. CS is the compressibility state (whether CF is 1, 2, or 4). When CF=1, only the CS field is used (*i.e.*, the block is not compressed).

3.3 Total Size-Related Overhead

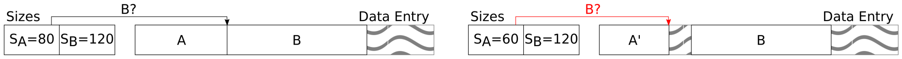
A data entry in a cache layout using Pairwise Space Sharing needs enough size-field bits to cover the worst-case scenario, in which the maximum amount of blocks compressed to the best compression factor ($maxCF$) are co-allocated (Figure 4). This means that besides the usual tag, replacement state and coherence fields, each data entry must dispose of $\log_2(maxCF)$ bits to inform the number of BPs in the data entry (equivalent to a conventional field informing the compressibility state — CS); and $\log_2(\frac{maxCF}{2})$ size field entries to bear the size of the smallest possible BP entry ($sizeBits_{maxCF}$), taking into account whether the single-size-per-BP optimization is being used — see Equation 4.

$$total_{entry} = \log_2(maxCF) + \log_2(\frac{maxCF}{2}) \cdot sizeBits_{maxCF} \cdot numSizesPerBP \tag{4}$$

For instance, the case of PSS where the maximum compressed size allowed is 50% is similar to constrained methods allowing three possible sizes — 25%, 50% and 100% — such as YACC [19] and SCC [18]: $maxCF = 4$, $BPSize_2 = 64B$, $BPSize_4 = 32B$, $maxSize_2 = 32B$, $maxSize_4 = 16B$, $sizeBits_2 = sizeBits_4 = 0$, thus $total_{entry} = 2$ bits.

3.4 Position Bits

It is important to notice that a block's size may not be the only piece of metadata needed to locate blocks: starting-position bits may also be necessary. When no positional information is explicitly provided the sizes are used to deduce the blocks' starting position. This means that a size can only be updated along with a recompaction step (see Figure 3.4). If the design chooses not to update a block's size right away on a data contraction or eviction, these bits must be added to the size of the previous block in the sequence so that the starting positions of the following blocks are kept correct - *i.e.*, those bits are wasted.



(a) A is stored at position 0, and B is stored relative to A.

(b) If A's size changes, and B is not moved, B can no longer be located.

Fig. 5. Locating blocks without a position field.

Under these circumstances, some layouts may opt to add a position field to the blocks' metadata. Assume that C is the number of coherence bits, up to four blocks can co-allocate in a data entry, and that the number of tag bits is the same for all methods: an unconstrained layout's metadata bits would include 1 compression-state (CS) bit per data entry, and 6 size bits and 6 position bits per block (Equation 5); a constrained layout needs 1/2 CS bits per data entry, and 1/2 index bits per valid sub-block (depends on the CS) (Equation 5); finally, PSS reduces the size restrictions of the constrained approach, so it adds one or two 3/4 size bits per valid block pair on top of T_{cons} (Equation 7).

$$T(otal)_{uncons} = 1 + 4 \cdot 6 \cdot 6 + 4 \cdot C \tag{5}$$

$$T_{cons} = \max(T_{consCF4}, T_{consCF2}) = \max(1 + 4 \cdot (2 + C), 2 + 2 \cdot (2 + C)) \quad (6)$$

$$T_{PSS} = \max(T_{consCF4} + 2 \cdot (1|2) \cdot 3, T_{consCF2} + 1 \cdot (1|2) \cdot 4) \quad (7)$$

For example, if C is 3, $T_{uncons} = 157$, $T_{cons} = 21$, and $T_{PSS} = 27|33$. As seen, **PSS has a huge metadata advantage over the unconstrained approach, while not adding much when compared to the constrained technique.**

4 Related Work

Dictionary-based compressors use the values in a cache line to fill a dictionary of previously seen values. While parsing a line, the dictionary entries are compared against patterns for full or partial matches, which are referred to by the compressed data, along with the discrepant bits in case of a partial match.

C-Pack [6] applies the basic ideas of dictionary-based compressors; X-Match and X-RL reorder the dictionary to apply Huffman code on the most-recently seen value [12]; BDI [15] limits the dictionary to two entries, which are matched through delta comparisons to achieve 1-cycle decompression; DISH [14] improves BDI's low compression efficiency by sharing dictionaries between multiple lines.

COCO [21] applies BDI's idea to objects, instead of cache lines; FPC-D [1] uses a 2-entry FIFO as its dictionary to reduce decompression latency. FPC [3] has no dictionary (*i.e.*, a pattern-only scheme); BPC [11] further compresses base-delta-like compressed data with bit-plane transformations; and SC^2 [4] uses probabilistic models to build a global dictionary.

Compressors must be associated with a compaction scheme to increase the effective cache capacity. Some compacted layouts allow any pair of lines to co-allocate by doubling the number of tags and informing the compressed sizes in the metadata [2,6,13]. This overhead is cumbersome, so modern proposals tend to focus on using *superblock tags*, which associate multiple neighbor blocks to a single shared tag [18,19]. Recent proposals move the tag information to the data entry [9]. Other approaches redesign the cache, with ideas ranging from adding extra caches holding the compressed data [8] to a full overhaul of the cache organization [21]. PSS is orthogonal to these design decisions.

Chen *et al.* introduced *pair-matching* [6], which co-allocates blocks in pairs as long as the sum of their compressed sizes' fits in a data entry, requiring one size field per sub-block. Pairwise Space Sharing has up to 73% less size-related metadata overhead due to its insights on the probabilities of co-allocation, and removal of the partially redundant companion's size information. Besides, since sub-block location is fixed, and unused bits are located in-between blocks, PSS greatly simplifies data insertion, removes the need for recompaction, and minimizes the likelihood of data expansions.

5 Methodology

Our simulations have been performed using gem5 [5]. Compression-related statistics are averaged across all (de)compressions. Compaction-related statistics are averages of snapshots (taken every 100K ticks) of the cache's contents. We took multiple checkpoints per benchmark of the SPEC 2017 benchmark suite [7] using SimPoints [20]. Workloads were warmed up for 100M instructions, and then executed for 200M instructions. The average of each benchmark's statistics has been calculated with the arithmetic mean of its checkpoints, and the total geometric mean of the benchmarks was normalized to a baseline system without compression. Benchmarks whose number of Misses Per Kilo-Instruction (MPKI) was lower than 1 were discarded from the analysis — as they barely benefit from having larger caches, compression is not useful.

The baseline model executes out-of-order (OOO), and is detailed in Table 1. All compression and compaction algorithms are applied to the L3 on top of this common configuration.

Cache line size	64B
L1 I/D	32KB, 4-ways, 4 cycles, LRU
L2	256KB, 8-ways, 12 cycles, RRIP [10]
Shared L3	1MB, 8-ways, 34 cycles, RRIP
MSHRs and write buffers	64

DRAM	DDR4 2400MHz 17-17-17, tRFC=350ns, 4GB
Processor	1 core, OOO, 8-issue
Architecture	ARM 64 bits
Clock	4GHz
Image	Ubuntu Trusty, Little Endian

Table 1. Baseline system configuration.

6 Results

In this section we analyse which size range provides the most cost-effective results. We also compare the efficiency and effectiveness of PSS when applied on top of multiple state-of-the-art compressors. Finally, we provide an area estimate comparison.

6.1 Selecting the Size-Field Range

As stated previously, neighbor blocks tend to have similar data contents, and thus similar compressibility. If a line compresses to a size greater than 50% of the BP's size, its companion has a high likelihood of compressing to a size greater than 50% too; consequently, there is lower co-allocation chance. This means that reducing size constraints may actually degrade performance.

To find out the best number of bits to be used in the size field, we have analysed the different compaction ratios achieved, as well as the proportion of blocks that co-allocated with another block at the moment it was compressed. *The highest ratio of successful co-allocations is achieved when block sizes are within the absolute range* [37.5%: 62.5%] *of the BP's size* (Figure 6, left). This is reflected in compaction ratio improvements (Figure 6, right). **Unless stated otherwise, future references to PSS use the** [37.5%: 62.5%] **range.**

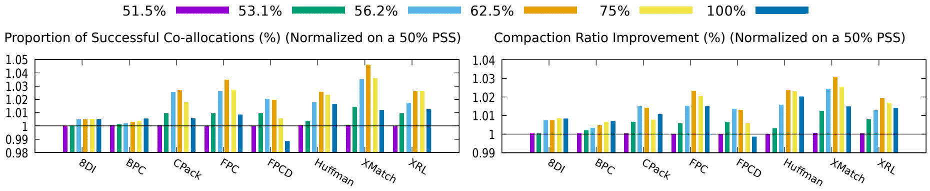


Fig. 6. Comparison of the best range choice for multiple state of the art compression methods under a YACC layout with PSS. Values for each compressor are normalized to the respective compressor using a PSS of 50%. Left plot is the ratio of successful co-allocations ($\frac{numCoAllocations}{numCompressions}$). Right plot is the compaction ratio.

6.2 Compaction Ratio

Figure 7 shows the difference in compaction ratio for various state-of-the-art compressors while coupling YACC [19] to PSS. *All PSS configurations outdo their non-PSS counterpart* because of the better ratio of successful co-allocations.

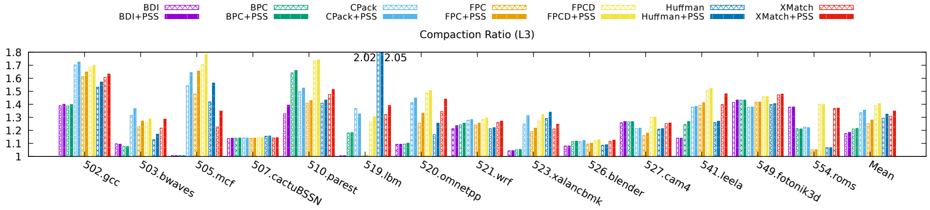


Fig. 7. Compaction ratio of multiple state of the art compression methods under a YACC layout without and with PSS applied to them. X-RL's results are similar to X-Match's, and are not shown due to space constraints.

6.3 Number of Size Fields per BP

Having a single size per BP has a drawback in case the block whose size is stored expands: the exact size of its companion is unknown, so the companion's data must be read, decompressed and re-compressed to check if evictions are needed. We simulated a worst case scenario where the latency of a read was added to every block overwrite, and the differences in IPC and compaction ratio were far below 1% because these steps could often be done off the critical path; hence, **halving the number of size fields has marginal negative impact**.

6.4 Effects on Data Expansions

Although rare, data expansions can be inconvenient. We compare the number of data expansions of YACC using PSS with 1) a conventional 50% constrained design (Con_{50}), and 2) a PSS design that co-allocates at non-extremities locations (*i.e.*, blocks are allocated sequentially, at the first available bit), at a byte granularity (PSS_{NE}). PSS generates a much smaller data-expansion footprint, as seen in Figure 8.

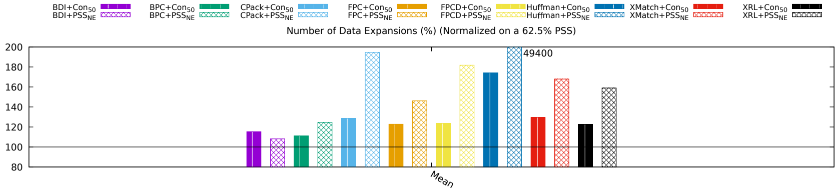


Fig. 8. Number of data expansions with Con_{50} and PSS_{NE} (normalized on PSS — *e.g.*, there are 15% more data expansions in BDI+Con_{50} than in PSS). Only the means of the benchmarks are shown due to space constraints. The number of data expansions is normalized on top of the selected PSS configuration, so any value above the horizontal line signifies that its respective configuration has more data expansions than PSS. All other configurations are above the horizontal line; thus, PSS has significantly less data expansions than any other compared configuration.

6.5 Area Overhead

We have implemented the placement decisioning logic under a generic unconstrained, a generic constrained, and the PSS approaches using Quartus II Web Edition v21.1 and assuming that the metadata contains a position field. Constrained and PSS require, respectively, 4% and 14% of the area of the unconstrained method. *These two techniques also manage to always calculate new positions in a single cycle, while the unconstrained approach needs multiple cycles at a much slower clock rate.*

7 Conclusion

This paper explores Pairwise Space Sharing (PSS), a special case of pairwise block compression which uses implicit information to reduce compression-size metadata, increase co-allocation opportunities, and remove re-compaction needs. This concept is layout-independent, but highly advantageous for spatially close block co-allocation techniques (*e.g.*, superblocks). PSS reaches an effective metadata-bits usage, and improves the compaction ratio of nearly every compressed-system configuration, while still being simple enough to handle compressed-line placement decision in a single cycle.

References

1. Alameldeen, A.R., Agarwal, R.: Opportunistic compression for direct-mapped dram caches. In: Proceedings of the International Symposium on Memory Systems.

p. 129–136. MEMSYS '18, Association for Computing Machinery, Alexandria, Virginia, USA (2018). https://doi.org/10.1145/3240302.3240429, https://doi.org/10.1145/3240302.3240429

2. Alameldeen, A.R., Wood, D.A.: Adaptive cache compression for high-performance processors. SIGARCH Comput. Archit. News **32**(2), 212 (Mar 2004). https://doi.org/10.1145/1028176.1006719, https://doi.org/10.1145/1028176.1006719
3. Alameldeen, A.R., Wood, D.A.: Frequent pattern compression: A significance-based compression scheme for l2 caches. Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep **1500** (2004)
4. Arelakis, A., Stenstrom, P.: Sc2: A statistical compression cache scheme. In: Proceeding of the 41st Annual International Symposium on Computer Architecuture. p. 145–156. ISCA '14, IEEE Press, Minneapolis, Minnesota, USA (2014). https://doi.org/10.1109/ISCA.2014.6853231, https://doi.org/10.1109/ISCA.2014.6853231
5. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. SIGARCH Comput. Archit. News **39**(2), 1–7 (Aug 2011). https://doi.org/10.1145/2024716.2024718, https://doi.org/10.1145/2024716.2024718
6. Chen, X., Yang, L., Dick, R.P., Shang, L., Lekatsas, H.: C-pack: A high-performance microprocessor cache compression algorithm. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on **18**(8), 1196–1208 (2010). https://doi.org/10.1109/TVLSI.2009.2020989, https://doi.org/10.1109/TVLSI.2009.2020989
7. Corporation, S.P.E.: Spec cpu 2017. https://www.spec.org/cpu2017/ (2017), accessed: 2019-10-10
8. Dusser, J., Piquet, T., Seznec, A.: Zero-content augmented caches. In: Proceedings of the 23rd International Conference on Supercomputing. p. 46–55. ICS '09, Association for Computing Machinery, Yorktown Heights, NY, USA (2009). https://doi.org/10.1145/1542275.1542288, https://doi.org/10.1145/1542275.1542288
9. Hong, S., Abali, B., Buyuktosunoglu, A., Healy, M.B., Nair, P.J.: Touché: Towards ideal and efficient cache compression by mitigating tag area overheads. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. p. 453–465. MICRO '52, Association for Computing Machinery, Columbus, OH, USA (2019). https://doi.org/10.1145/3352460.3358281, https://doi.org/10.1145/3352460.3358281
10. Jaleel, A., Theobald, K.B., Steely, S.C., Emer, J.: High performance cache replacement using re-reference interval prediction (rrip). In: Proceedings of the 37th Annual International Symposium on Computer Architecture. p. 60–71. ISCA '10, Association for Computing Machinery, Saint-Malo, France (2010). https://doi.org/10.1145/1815961.1815971, https://doi.org/10.1145/1815961.1815971
11. Kim, J., Sullivan, M., Choukse, E., Erez, M.: Bit-plane compression: Transforming data for better compression in many-core architectures. SIGARCH Comput. Archit. News **44**(3), 329–340 (Jun 2016). https://doi.org/10.1145/3007787.3001172, http://doi.acm.org/10.1145/3007787.3001172
12. Kjelso, M., Gooch, M., Jones, S.: Design and performance of a main memory hardware data compressor. In: EUROMICRO 96. Beyond 2000: Hardware

and Software Design Strategies., Proceedings of the 22nd EUROMICRO Conference. pp. 423–430. IEEE, IEEE Computer Society, Prague, Czech Republic (1996). https://doi.org/10.1109/EURMIC.1996.546466, https://doi.org/10.1109/EURMIC.1996.546466

13. Lee, J.S., Hong, W.K., Kim, S.D.: Design and evaluation of a selective compressed memory system. In: Computer Design, 1999.(ICCD'99) International Conference on. pp. 184–191. IEEE, IEEE Computer Society, Austin, Texas, USA (1999). https://doi.org/10.1109/ICCD.1999.808424, https://doi.org/10.1109/ICCD.1999.808424
14. Panda, B., Seznec, A.: Dictionary sharing: An efficient cache compression scheme for compressed caches. In: The 49th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-49, IEEE Press, Taipei, Taiwan (2016). https://doi.org/10.1109/MICRO.2016.7783704, https://doi.org/10.1109/MICRO.2016.7783704
15. Pekhimenko, G., Seshadri, V., Mutlu, O., Gibbons, P.B., Kozuch, M.A., Mowry, T.C.: Base-delta-immediate compression: Practical data compression for on-chip caches. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques. p. 377–388. PACT '12, Association for Computing Machinery, Minneapolis, Minnesota, USA (2012). https://doi.org/10.1145/2370816.2370870, https://doi.org/10.1145/2370816.2370870
16. Salomon, D.: Data Compression: The Complete Reference, 3rd Edition. Springer (2004), http://www.davidsalomon.name/DC3advertis/DComp3Ad.html
17. Sardashti, S., Arelakis, A., Stenström, P., Wood, D.A.: A primer on compression in the memory hierarchy. Synthesis Lectures on Computer Architecture **10**(5), 1–86 (2015). https://doi.org/10.2200/S00683ED1V01Y201511CAC036, https://doi.org/10.2200/S00683ED1V01Y201511CAC036
18. Sardashti, S., Seznec, A., Wood, D.A.: Skewed compressed caches. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. p. 331–342. MICRO-47, IEEE Computer Society, Cambridge, United Kingdom (2014). https://doi.org/10.1109/MICRO.2014.41, https://doi.org/10.1109/MICRO.2014.41
19. Sardashti, S., Seznec, A., Wood, D.A.: Yet another compressed cache: A low-cost yet effective compressed cache. ACM Trans. Archit. Code Optim. **13**(3) (Sep 2016). https://doi.org/10.1145/2976740, https://doi.org/10.1145/2976740
20. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. In: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems. p. 45–57. ASPLOS X, Association for Computing Machinery, San Jose, California (2002). https://doi.org/10.1145/605397.605403, https://doi.org/10.1145/605397.605403
21. Tsai, P.A., Sanchez, D.: Compress objects, not cache lines: An object-based compressed memory hierarchy. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 229–242. ASPLOS '19, ACM, Providence, RI, USA (2019). https://doi.org/10.1145/3297858.3304006, http://doi.acm.org/10.1145/3297858.3304006