CHALMERS
UNIVERSITY OF TECHNOLOGY

FlatPack: Flexible Compaction of Compressed Memory

FlatPack: Flexible Compaction of Compressed Memory

Albin Eldstål-Ahrens
eldstal@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

Angelos Arelakis
angelos.arelakis@zptcorp.com
ZeroPoint Technologies AB
Gothenburg, Sweden

Ioannis Sourdis
sourdis@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

Abstract

The capacity and bandwidth of main memory is an increasingly important factor in computer system performance. Memory compression and compaction have been combined to increase effective capacity and reduce costly page faults. However, existing systems typically maintain compaction at the expense of bandwidth. One major cause of extra traffic in such systems is page overflows, which occur when data compressibility degrades and compressed pages must be reorganized. This paper introduces FlatPack, a novel approach to memory compaction which is able to mitigate this overhead by reorganizing compressed data dynamically with less data movement. Reorganization is carried out by an addition to the memory controller, without intervention from software. FlatPack is able to maintain memory capacity competitive with current state-of-the-art memory compression designs, while reducing mean memory traffic by up to 67%. This yields average improvements in performance and total system energy consumption over existing memory compression solutions of 31-46% and 11-25%, respectively. In total, FlatPack improves on baseline performance and energy consumption by 108% and 40%, respectively, in a single-core system, and 83% and 23%, respectively, in a multi-core system.

CCS Concepts

• **Computer systems organization** → *Other architectures*; • **Hardware** → *Memory and dense storage*.

Keywords

Memory system, Memory compression

ACM Reference Format:
Albin Eldstål-Ahrens, Angelos Arelakis, and Ioannis Sourdis. 2022. FlatPack: Flexible Compaction of Compressed Memory. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '22), October 10–12, 2022, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3559009.3569653

1 INTRODUCTION

Main memory is a critical resource in modern systems. Its capacity must be sufficient to avoid frequent page faults and its bandwidth high enough to accommodate the rates of requested data. The demand for both memory capacity and memory bandwidth is increasing as applications become more data-intensive and a larger number of cores is integrated on a single chip. However, simply scaling up memory size and bandwidth increases system cost and power consumption [20].

Data compression has the potential to offer a better cost to performance tradeoff in computing systems by more efficiently utilizing the capacity and bandwidth resources of main memory. Previous techniques are able to achieve improvements in either capacity or bandwidth but usually not in both. Some designs use *memory compression* to reduce memory traffic without considering capacity improvement [10, 12]. Others aim primarily at *memory compaction* to increase storage density and effective capacity [6, 28], exploit free prefetching effects [28, 39], but introduce significant traffic overheads to manage compacted memory [6].

There are several reasons for the traffic overheads of managing a compressed and compacted memory. First, compressed blocks require additional *metadata* to be accessed. Second, compressed blocks have variable size and therefore may cross the boundaries of a regular memory location requiring two *split accesses*. Even if they do not span across the access boundaries, writing a compressed block back to memory often requires a *read-modify-write (RMW)* operation in order to preserve data of neighboring blocks. Another source of traffic overheads and inefficiency is the *change in compressibility of data* and hence in their size during execution, i.e., blocks growing (or even shrinking) during runtime. This variation in compressibility leads to inefficiencies in existing compaction systems, which compact blocks into sequential spaces of rigid size. A growing block may be stored uncompressed as an *exception*, in space specially reserved for this purpose [6, 28]. Alternatively, the change causes a *page overflow* which requires the entire page to be brought on-chip, repacked and migrated, inducing a memory traffic overhead. In our experiments, an average of 15% of cache line updates lead to an increase in compressed size, causing an *exception* at the expense of memory capacity and traffic.

Although some of the above issues, e.g. metadata [16], have been addressed in the past, the bandwidth benefits - if any - of current state of the art memory compaction designs are far from the achieved raw compression ratio. As illustrated in Figure 1, the required memory traffic of such designs is comparable to the traffic of a baseline with no compression, 2×-4× higher than the theoretical minimum indicated by the achieved compression ratio. In effect, existing memory compaction techniques consume significant bandwidth and so limit system performance and energy efficiency.

This work introduces FlatPack, a novel technique aimed at reducing the memory bandwidth overheads of memory compaction. The key observation behind this work is that existing systems are unable to efficiently handle dynamically varying compressibility of data, which leads to excessive page overflows and hence excessive memory traffic. FlatPack mitigates this problem by allowing compressed blocks to be fragmented within a physical page and share expansion space providing the flexibility to be reorganized independently

PACT '22, October 10–12, 2022, Chicago, IL, USA

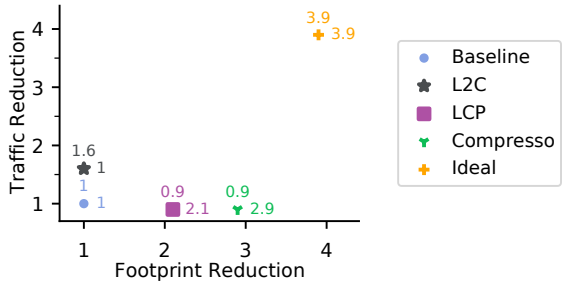


Figure 1: State-of-the-art memory compression systems, classified by their traffic reduction and footprint reduction. *Ideal* shows the achievable compression, *LCP* and *Compresso* are compacting systems, L^2C is a non-compacting system.

without disturbing other blocks. FlatPack's flexible reorganization can be performed in response to all size changes without introducing additional data movement. Furthermore, by fragmenting blocks at the Memory Access Granularity (MAG), FlatPack reduces RMW traffic, since most compressed memory writes only affect one block. FlatPack's reorganization is performed in hardware by the memory controller, without intervention by system software or the operating system. In effect, FlatPack's flexibility to handle variability in block size reduces data movement and memory traffic improving system performance and energy efficiency.

Concisely, FlatPack is a novel flexible memory compaction approach that aims to reduce the traffic overheads and makes the following contributions:

- a flexible format of compressed pages that allows compressed blocks to be fragmented and share expansion space in the physical memory region of the page offering efficient memory compaction;
- a hardware mechanism that enables the memory controller to exploit the above format and dynamically reorganize data within the page, without software intervention and with minimal data movement;
- a thorough evaluation of FlatPack and comparison with current state of the art memory compaction designs to measure the significant reduction in memory traffic and impact of FlatPack in system performance and energy.

The remainder of this paper is organized as follows. Section 2 discusses background and related work. Section 3 describes the proposed FlatPack architecture. Section 4 presents evaluation results and Section 5 draws our conclusions.

2 BACKGROUND AND RELATED WORK

A number of systems have been proposed to compress and compact main memory, to save bandwidth or increase memory capacity. Any memory compression or compaction design is subject to a number of design choices, which are detailed below. The two designs currently at the forefront of memory compaction are Linearly Compressed Pages (LCP) [28] and Compresso [6]. This section outlines the design parameters of these and other related approaches, as well as a summary of the design choices employed by FlatPack.

2.1 Compression Algorithm

A number of algorithms have been proposed for compression in the memory hierarchy. The primary requirement for a suitable compression algorithm is low decompression latency, to minimize performance impact. Lossless compression schemes typically offer compression ratios up to 2× to 4× on real-world data [1–3, 5, 18, 29]. For applications which tolerate approximation, lossy compression offers more aggressive compression ratios of up to 16× [10, 12, 30] or allows for bandwidth optimizations in combination with lossless compression [19]. Deduplication has been proposed as an alternative to compression [27]. The more complex Lempel-Ziv algorithm has also been employed, using an additional cache to hide its decompression latency [35].

FlatPack is compatible with any block compression algorithm, without loss of generality. The system is evaluated here with the SC^2 compression scheme, which offers a competitive compression ratio and low-latency operation [3].

2.2 Compression Granularity

One way to differentiate memory compression systems is based on compression granularity, i.e. the size of the data block being compressed as one unit. Two basic approaches are possible, each with different characteristics. *1) Compressing individual cache lines* at the granularity of the Last Level Cache (LLC) [4, 6, 9, 26–28, 30, 34, 40]. The benefit of this is that there is no overhead from fetching unused compressed data. On the other hand, general purpose DRAM is restricted to a minimum Memory Access Granularity (MAG), which is typically tuned to be the size of one cache line. As a result, single cache line compression has limited potential for bandwidth reduction. Similarly, the MAG prevents individual, smaller than MAG, compressed blocks from being updated in memory, forcing the memory controller to perform a Read-Modify-Write (RMW) sequence. Instead of fixing the compressed block size to the size of the cache line, another alternative is to increase the cache lines to the size of the compressed block. Then, the limitations to the potential bandwidth reduction can be overcome, but excessively large cache lines and a potentially large LLC are required. One such example is MXT [35], which uses very large compressed blocks and cache lines (1 KB each) requiring a large off-chip LLC.*2) Compressing multiple cache lines* together, making up a larger compression block [7, 10, 12, 15, 17, 19, 32, 35]. The benefit of this is that the compressed block may exceed the MAG, and thus memory transfers are more efficiently utilized. Conversely, these systems do not support random access of individual cache lines within a compressed block. This complicates writebacks to memory and enforces a form of prefetching of all co-compressed cache lines.

LCP and Compresso both compress single cache lines. FlatPack compresses blocks of four cache lines, in order to be able to improve memory bandwidth utilization and reduce the need for RMW operations. In addition, the larger compressed blocks are key to enabling a flexible approach to block compaction.

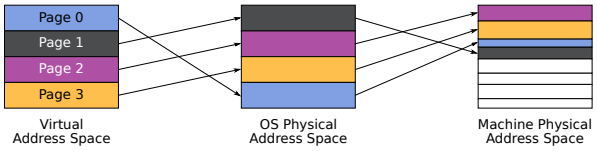


Figure 2: The three address spaces used for memory compaction. In a regular system, OSPA space is the physical space.

2.3 Block Compaction

While memory compression may give a bandwidth benefit, capacity improvement requires compressed blocks to be *compacted* in physical memory. Some systems forgo compaction altogether, aiming only to improve bandwidth utilization [10, 12, 17, 19, 30, 34] or to make space for error correcting codes [26]. Several compaction approaches have been proposed in literature. *1) LCP packing* is the simplest form of compaction, assigning an identically sized space for each block within a page [7, 28, 40]. This simplifies address calculation, at the expense of wasted space for blocks with higher compressibility. *2) Line Packing* is employed by Compresso and others, packing the compressed blocks of a page together while supporting more than one block size [4, 6, 9, 32]. This allows for greater benefit when compressibility varies, eliminating more wasted space. *3) Block Fragmentation* as employed by MXT and related designs compresses very large blocks (1kB) and fragment compressed blocks in fixed-size sectors (256 KB) in memory without limitation to their page sharing relationships [15, 35]. Such free placement yields a large number of physical memory addresses stored as metadata, which can be a significant overhead. MXT's high metadata overhead is kept in check using coarse fragment granularity (128B half-sectors), which however leads to compaction inefficiencies because any data smaller than the fragment size wastes memory capacity. *4) BCD Deduplication* divides physical memory into several large arrays, each storing a different part of all compressed blocks [27].

FlatPack allocates a contiguous space for each logical page, thus keeping a single physical address in its metadata. Compressed blocks are fragmented at MAG granularity and placed freely within that space. The fragmentation and location of compressed blocks is dynamic, automatically adapting to the changing compressibility of data. Furthermore, it allows blocks within a page to differ in size. This automatic reorganization of blocks is performed by the memory controller, in hardware. Since it occurs only on block write-back, blocks are reorganized without any additional data movement. FlatPack stores the first part of each compressed block in a fixed location within the page, which allows memory access to begin in parallel with further address calculation for the compressed block.

2.4 Address Translation and Page Compaction

A standard, uncompressed memory hierarchy has one level of address translation. The *Virtual Address* (VA) space of each process is translated into the *Operating System Physical Address* (OSPA) space, where pages are the same size and uncompacted. Any memory compaction system must modify or supplement the address translation by introducing an additional *Machine Physical Address* (MPA) space, as illustrated in Figure 2. This is a necessity to support blocks or pages of non-uniform size and gain memory capacity.

The implementation of this additional MPA space depends on the exact organization of compacted memory. Two principal approaches are taken by existing systems, illustrated in Figure 3. *1) Contiguous pages* are used by LCP and others, allocating contiguous memory space for a single compressed page, smaller than the system's normal page size [7, 9, 28]. This approach requires only one MPA to locate each page, but relies on a relatively static compressibility. Reduced compressibility can cause page overflows, requiring the full compressed page to be migrated to a larger allocation. *2) Fragmented pages* is an alternative approach, where a compressed page is allocated as a set of smaller chunks. The benefit of this organization is the ability to dynamically append chunks of physical memory to a given compressed page [4, 6, 40]. Employed by Compresso and others, this approach is better able to deal with compressibility changes, but requires more metadata to locate multiple chunks in physical memory. An extreme example of this alternative is MXT [35], where each compressed block fragment (sector) is free to be placed anywhere in memory. Besides the high metadata overhead, this significantly complicates address translation adding latency overheads.

FlatPack uses fixed contiguous compressed pages for their simplicity and reduced metadata. Compressibility changes are handled by improved flexibility in block placement within each compressed page. By allowing blocks to grow and shrink, FlatPack reduces the number of page overflows.

2.5 Metadata Handling

All compression schemes require additional metadata to manage compressed blocks and pages. This metadata describes block sizes, compression methods, placement in physical memory and other auxiliary compression information which is not application data. A number of approaches exist for organization and transfer of compression metadata. *1) Metadata in main memory, separated from the compressed data* is the most common organization [6, 7, 15, 17, 27, 30, 35, 40]. An on-chip metadata cache is typically employed to reduce the traffic and latency overhead of finding the metadata for accessed pages. To further reduce latency, the metadata table can be accessed in parallel with the page table [9, 10, 12, 32]. *2) Metadata embedded in the compressed block* has been proposed as a method to reduce the bandwidth overhead of metadata traffic [16, 19, 26, 34]. This is unsuitable for memory compaction, as it decouples the page-level metadata from some of the blocks within the page. *3) Metadata embedded in the compressed page* adds this ability, and avoids fragmenting the physical address space [4, 28].

Compresso uses a separate metadata table accessed on demand, LCP embeds metadata in its compressed pages. Both designs employ a small metadata cache.

FlatPack uses a separate metadata table, accessed in parallel with the page table. An on-chip metadata cache is kept updated in tandem with the TLB. This approach guarantees that when requests reach the LLC or memory controller, the block and page compression metadata is available on-chip.

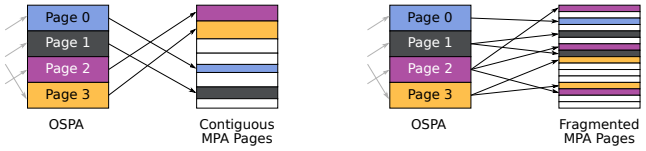


Figure 3: The two principal approaches to page compaction. Contiguous pages require a single MPA, fragmented pages may require multiple.

2.6 Last-Level Cache Support

A number of memory compression systems also modify or use the LLC in ways designed to optimize or support memory compression. One approach is to unify LLC and main memory compression by storing only compressed blocks in the cache [15, 27]. This increases the effective capacity of the LLC, without additional SRAM cost. Another approach is to allow the LLC to store compressed blocks alongside regular uncompressed cache lines [10, 12, 17]. This allows large-block memory compression schemes to offset the traffic overhead of reading large blocks from memory and benefit from spatial locality. Compression systems which compress at the granularity of single cache lines may pack multiple of these together, in order to satisfy the memory access granularity. Such extra cache lines can be decompressed into an otherwise unmodified LLC [6, 28] or a small special-purpose cache [4] in order to gain some bandwidth benefits. LCP and Compresso both work with an unmodified LLC, and insert all valid overfetched data. MXT adds a large off-chip cache level in order to hide the decompression latency of its large blocks and support the required large cache lines [35].

FlatPack compresses blocks consisting of multiple MAGs of data. To mitigate the traffic overhead of LLC misses, it employs a modified Decoupled Sectored Cache [33] to co-locate both compressed blocks and uncompressed cache lines in the LLC similarly to AVR, MemSZ, and L^{2}C [10–12].

2.7 Overheads of Existing Systems

Current state of the art memory compaction systems, Compresso [6] and LCP[28], suffer from two primary types of overhead. First, rigid and static assignment of compressed space for each block leads to deteriorating compaction as compressibility changes. While Compresso allows for more than one block size per page, it is unable to handle blocks changing size over time. Growing blocks are left uncompressed while shrinking blocks continue occupying their initially assigned space. Over time, this deterioration leads to a page overflow forcing the system to recompress and recompact the page. The second major source of overhead is the compression granularity of single cache lines. Compresso and LCP both fetch individual compressed cache lines from memory on an LLC miss, which is rounded up to the memory access granularity (MAG). As a result, one full MAG is transferred from memory for each cache line, regardless of compressibility. In some cases, this overfetching can be beneficial, if it contains additional complete compressed cache lines. This gives a modest prefetching effect, which can offset part of the bandwidth overhead.

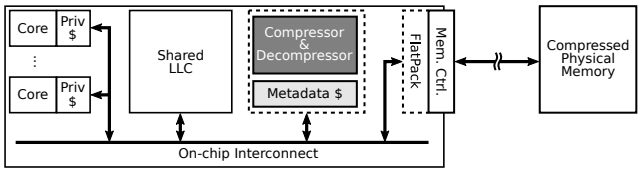


Figure 4: The FlatPack memory system utilizes a modified LLC and adds compressor hardware and a metadata cache.

Another design with different design choices is MXT [35]. MXT does not suffer from the first aforementioned overhead as it offers flexibility to place fragments of compressed blocks anywhere in memory, allowing shrinking and expansion without huge traffic overheads. However, this flexibility in MXT has a high price as it complicates address translation adding latency and causes metadata inefficiencies requiring coarse granularity of compressed blocks and block fragments affecting compaction efficiency. Moreover, MXT suffers from the second aforementioned overhead in a different way. Due to its metadata inefficiency and large compressed blocks, it is forced to use a large LLC with large cache lines.

FlatPack mitigates capacity waste by compressing larger blocks and fragmenting them in physical memory. Each page is divided into *slots* at the memory access granularity. Each compressed block is fragmented into MAG-sized parts and a number of slots are assigned to it. When a block's compressed size changes, slots can be released and reassigned as needed, without additional data movement overhead. This maintains compaction efficiency, by dynamically adapting space assignment to each block. FlatPack offers this flexibility in a metadata-efficient manner, supporting fine compressed block granularity while avoiding MXT's LLC, translation and compaction inefficiencies.

3 FLATPACK DESIGN

Currently published state-of-the-art memory compression systems primarily target one of two bottlenecks of main memory: memory bandwidth or memory capacity. Systems which maintain data compacted in main memory do so at the expense of additional memory traffic [6, 28]. This is mainly due to *data movement*, e.g., page migration required to reorganize a compressed page in order to eliminate fragmentation and handle varying compressibility. Conversely, the most straight-forward compression approach to reducing memory traffic leaves memory capacity unimproved [10, 17, 34]. This eliminates the bandwidth overhead of page reorganization, and simplifies address translation logic.

FlatPack aims to combine these approaches in order to gain the benefits of both. Using a novel compaction scheme, FlatPack offers memory capacity improvements on par with current state-of-the-art systems. By being flexible to compressibility changes, as well as allowing size skew between blocks, FlatPack is able to dynamically reorganize physical memory on demand with fewer costly page migrations.

The basis of this flexibility is the fragmentation of compressed blocks, at the Memory Access Granularity (MAG) dictated by the physical memory controller. MAG-sized *slots* in physical memory are assigned and reassigned on demand, as blocks shrink and grow.

By compressing blocks which are larger than the MAG, the compressed data can be fragmented and flexibly placed. This allows a compressed page to support a wide variety of block sizes, and allows blocks to change size over time as long as the average compressibility across the page does not increase. Crucially, blocks are able to change size independently of each other. Reorganization of a block is performed on demand whenever the block is written back to memory, and thus introduces no additional data movement.

Figure 4 shows a top-level overview of the FlatPack architecture. A specialized compressor and decompressor hardware module is added, as well as a small on-chip metadata cache. The shared Last-Level Cache (LLC) is designed to store both uncompressed cache lines and complete compressed blocks, using a Decoupled Sectored Cache organization [33]. The flexible packing and organization is performed in hardware by a module situated on the core side of the Memory Controller. This module implements a Finite State Machine which receives FlatPack block and page operations from the LLC and uses metadata to translate them into individual requests to the standard memory controller. Allocation of physical memory on the page level is performed by a software runtime.

Similarly to other memory compaction systems, FlatPack introduces an additional layer of address translation. Virtual addresses are translated using a Page Table into the OS Physical Address (OSPA) space. OSPA is used for cache tags, but does not represent physical memory. FlatPack organizes data in the Machine Physical Address (MPA) space, using variable-size compressed pages. In MPA space, compressed pages are compacted to maximize the available memory capacity.

3.1 Compression

The main feature of FlatPack is the ability to dynamically pack and repack compressed blocks in physical memory on demand. To avoid bandwidth waste, compaction is performed at Memory Access Granularity (MAG), which is typically 64 bytes and equal to an LLC cache line. Lossless compression typically offers between 2× and 4× compression ratio on real-world data [1–3, 5, 18, 29]. In order to generate compressed blocks which can be fragmented into multiple MAGs, FlatPack compresses blocks of 256B into compressed sizes between 32B (1/2 MAG) and 256B. This gives a theoretical maximum compression ratio of 8×.

FlatPack is compatible with any block compression algorithm able to compress 256 bytes of data. In this paper, we evaluate using SC^2 [3]. SC^2 uses a global Value Frequency Table (VFT) to assign shorter encodings to frequently appearing bit sequences. The VFT content is created dynamically by profiling a small random part of memory, while encoding transition is implemented using a similar mechanism presented in the original SC^2 work. Originally designed for cache compression, SC^2 exhibits high compression ratio for real-life datasets, as well as low-latency compression and decompression. These properties make it suitable for memory compression as well.

3.2 Last-Level Cache

The use of multi-MAG memory blocks introduces a potential bandwidth overhead from overfetching. Compressing multiple neighboring cache lines as one block before writing to memory requires the block to always be transferred in its entirety. Reading multiple MAGs worth of compressed data from memory to serve a single LLC miss trivially leads to additional data transfer(s) compared to an uncompressed system. One way to alleviate this is to store the compressed block on-chip, exploiting spatial locality to allow the same block to serve future LLC misses. Similarly to previous works [10, 12], FlatPack employs a *Decoupled Sectored Cache* (DSC) for this purpose [33]. A DSC decouples the cache tags from the data, allowing multiple cache lines to share a tag. This is accomplished by introducing an array of Back Pointers, each associating a data entry with its tag as illustrated in Figure 5. Our implementation is extended with the ability to store both uncompressed single cache lines and compressed blocks, co-located in the same data array similarly to AVR and MemSZ [10, 12]. The main benefit of this co-location is that LLC requests can be served either using uncompressed data (like a regular set-associative LLC) or compressed data, adding decompression in order to avoid a costly memory access.

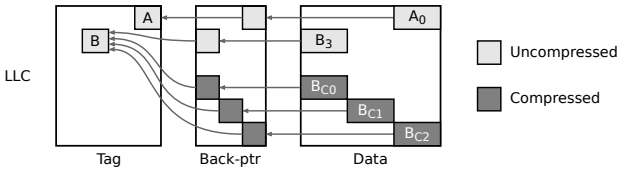


Figure 5: The Decoupled Sectored Cache design employed by FlatPack. The uncompressed cache lines B_x share a tag with the compressed version of the block stored in B_{Cy}.

3.3 Lazy Evictions

Another challenge introduced by large memory blocks is updates. As the MAGs within a block are compacted completely, there is no ability to update a single compressed cache line in memory without also updating the full block. As a result, writebacks of dirty lines from LLC directly into compressed memory cannot be performed. In order to recompress a block which is not available on chip, the full compressed block must be read from memory. *Lazy Evictions* mitigate this overhead by delaying the actual recompression using the empty space left in memory by compressed blocks [10]. If there is free space in physical memory, the dirty cache line is written back uncompressed and the block's metadata is updated to reflect this. The next time the block is fetched from memory, all lazily evicted cache lines are also read, and the block is recompressed to include the dirty data. The end result is that the dirty cache line is written to memory once and read from memory once, which is a smaller traffic overhead compared to reading the full compressed block for recompression and then writing it back to memory. In other words, lazy evictions are employed to reduce the traffic overhead of evictions. If a compressed block is not available on-chip, but a dirty line belonging to it is evicted, the compressed block must be updated. Without lazy evictions, a *read-decompress-modify-recompress-write* operation would be necessary, at a traffic cost. For a block of N MAGs, this will incur N reads and a similar number of writes. A lazy eviction, on the other hand, writes the dirty cache line back to memory uncompressed. The next time the block is read, the dirty line is also read. The total traffic caused by the lazy eviction is one write and one read.

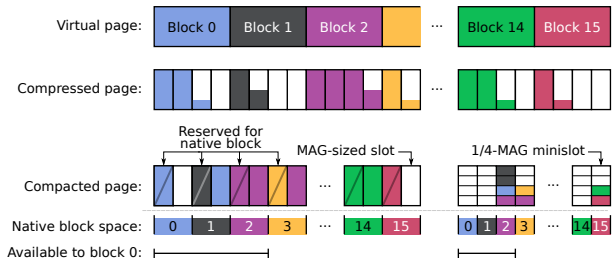


Figure 6: The layout of a compacted physical page, compared to an uncompacted compressed page. Slots native to a block N are also available to neighboring blocks N-1 and N-2.

3.4 Block Compaction

The variable size of compressed memory blocks necessitates a dynamic method for compaction in physical memory. Assigning a fixed space to each block risks introducing fragmentation as the block changes size. A shrinking block will leave parts of its space unused. A growing block must be relocated, leaving its original space unused. FlatPack breaks this fixed structure by dividing physical space into MAG-sized *slots* and dynamically assigning one or more slots to each block when needed. As blocks shrink and grow over time, their slot allotments change as needed, while unused slots remain available for other blocks within the same page. Crucially, a compressed block which changes size can be moved without affecting other blocks, and thus without additional data movement.

Figure 6 shows a FlatPack-compressed page of 16 compressible blocks. The compressed page occupies a section of physical memory, of a fixed size. This allocation is divided into MAG-sized *slots*. Each logical block is assigned an equal number of slots where it is *native*. The first of these slots is reserved for the native block itself. Any other slot native to a block N is made available to store the block N or its neighboring blocks $N-1$ and $N-2$ in a circular fashion. For example, parts of block 14 may be stored in a memory slot native to blocks 14, 15 or 0.

By compressing blocks of four MAGs, FlatPack generates compressed blocks which may exceed a single MAG. By dividing the compressed block into MAG-sized pieces, placement is flexible within the block's native space and that of the two following blocks. When a block is written to memory, on-chip metadata allows selection of a suitable slot set for the block. These properties allow a page's physical allocation to be reorganized on the fly without additional data movement.

Another advantage of this organization is that it supports pages where block sizes are non-uniform and variable. A FlatPack compacted page with a fixed size is able to support a wide variety of block size combinations, as well as handling compressibility variation over time. As long as the average compressibility across the entire page remains stable, the memory controller is able to manage individual blocks growing and shrinking without incurring additional reorganization traffic or intervention by system software.

3.5 Minislots

Dividing blocks along the memory access granularity introduces one important drawback, as all compressed sizes are effectively rounded up to multiples of the MAG. In the worst case, this means over-using and over-transferring (MAG-1) bytes per compressed block, wasting memory capacity and bandwidth on the order of one MAG per block. A majority of blocks will have some amount of such *granularity waste*, as it is unlikely that the compressed size will be an exact multiple of the MAG. FlatPack mitigates this problem by supporting finer granularity in a subset of its slots in physical memory. Depending on the allocated physical page size, each block is allotted two or four *minislots*. Each minislot is one quarter of the size of a regular slot, and made available to neighboring blocks in exactly the same way as regular slots. Figure 6 illustrates a compressed page with four minislots native to each block. Block 0 has compressed to two full-sized slots and the remainder fit within a single minislot. Without minislot support, the block would occupy three full-sized slots, with ¾ slot granularity waste. By prioritizing minislots in the same MAG as existing data, free minislots are also concentrated, reducing fragmentation. Contrary to full-sized slots, updating a minislot may require a read-modify-write operation.

3.6 Slot Assignment

When a block is evicted from the last-level cache, it must be written back to main memory. Due to data being updated, the block may have changed its size. As a result, physical packing of compressed blocks is updated on block writeback.

As explained in Section 3.4, the space of a compressed page in physical memory is divided into *slots*, with each slot being assigned to at most one block. Any given block is able to make use of slots from three separate locations: its own native space, as well as the respective native spaces of the following two blocks. For example, block 5 of a page is able to use the slots native to blocks 5, 6, and 7. The very first slot native to block 5 is reserved for that block, and thus always contains the first part of the block's data. Successive MAG-sized parts of the block are placed greedily in the first available slots. The fixed use of the first slot allows memory access to begin immediately on a cache miss, in parallel with address calculation for any remaining data.

After placing all full MAG-sized parts of the block in full-sized slots, a remainder is likely to exist which requires less than a full slot of physical space. To reduce granularity waste, FlatPack attempts to place this remainder data in one or more of the *minislots* reserved at the end of the page. Similarly to the full-sized slots, each minislot has one *native* block, and is made available to the native block and the two preceding ones. Placement of remainder data in minislots prioritizes data packing, to leave as many slots as possible unoccupied. Since the block is packed and written to memory in its entirety, there is no need for additional metadata to maintain the order of data within a block; slot are filled in their logical order.

3.7 Page Compaction

In order to maximize capacity gains, a memory compaction system is designed to minimize the allocated space for any given page. FlatPack uses fixed-size MPA pages with a set of predefined sizes organized in accordance with Table 1. A pool of allocated and free pages for each size is maintained by a software runtime. Each

Table 1: FlatPack compressed page sizes and organization.

Page Size	512B	1kB	1.5kB	2kB	2.5kB	3kB	3.5kB	4kB
Full Slots Per Block	0 0	0 0	16 1	16 1	32 2	32 2	48 3	64 4
Minislots Per Block	32 2	64 4	32 2	64 4	32 2	64 4	32 2	0 0

OSPA page is assigned one MPA page from one of these pools, and the assignment is stored in a separate metadata table, leaving the OSPA assignment in the regular page table. As a result, OS address translation and cache tagging logic remain unmodified.

The assignment of MAG-sized slots to native blocks sets the lower bound of a page's size to one slot per block. Because an uncompressed block fits in 4 MAGs, this would limit compression ratio to 4×, and would leave no block size flexibility at that size. To counteract these problems, FlatPack increases granularity for very small pages and introduces a 512 byte page with a compression ratio of 8×. Compressed pages of 512B or 1kB have no full-sized slots, and are instead entirely composed of minislots. In a 512B page, each block is native to two minislots. This finer granularity allows small pages to remain flexible to block size variation.

One drawback of fixed-size compressed pages is *Page Migration*. Page migrations occur when a compressed page exceeds its allocated size (page overflow) or is deemed able to fit in a smaller size (page shrink). To minimize the number of page size changes, FlatPack employs a runtime mechanism to estimate the page's future size at the time of initial allocation. Page allocation occurs in two stages. Initially, a new page is introduced into the page table by the operating system, and mapped to a read-only zero-data page. On the first writeback from the LLC (modifying the data in memory) a unique compressed page is allocated in MPA space. The size of this allocation is based on the compressibility of the written-back data. The compressed size of that data is extrapolated to the full page's size. If this expected page size exceeds one slot per block, a margin is added for flexibility. Finally, the estimate is rounded up to the nearest supported page size and used for the initial allocation.

3.8 Interaction with the OS

FlatPack uses memory ballooning in order to handle the variable memory size due to compression. Ballooning is a common feature in virtualization environments [37], and has been used for compressed memory by IBM [13] and more recently in Compresso [6].

In a virtualization environment, the Hypervisor controls the amount of available memory to the guest OS through a memory balloon software driver implemented in every guest OS. In essence, the Hypervisor can use the balloon driver to reclaim memory from one guest OS and provide it to others depending on the runtime memory needs of the respective virtual machines. The reclaimed memory from the respective guest OS is reserved by the balloon driver and cannot be allocated by the OS itself.

In a compressed memory system, the OS starts with a memory size $M = C \times P$, for maximum compression ratio C and physical capacity P. In the beginning of execution the memory is still uncompressed, thus the overcommitted memory $(C-1) \times P$ is reserved by the memory ballooning software driver. As the system allocates memory, the FlatPack runtime compresses the data and manages

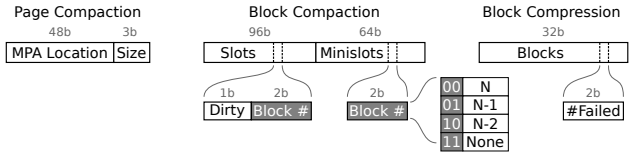


Figure 7: Metadata for a compacted FlatPack page. Two bits are used to encode which block occupies any given slot.

pools of allocated and free memory pages. New free memory pages are released to the OS by *deflating* the balloon driver. This way, free memory pages can be directly allocated by the OS. If memory compressibility deteriorates due to page overflows, the balloon driver is *inflated*. This triggers the OS's memory reclamation to free up allocated memory using the paging mechanism. The reclaimed addresses are communicated to the FlatPack runtime system, freeing the corresponding physical memory. If the system is under high memory pressure and the expanded, due to compression, memory has been exhausted the ballooning mechanism will be used again. The balloon driver of the guest OS will be inflated triggering page reclamation and causing possibly some pages to be swapped out. This will further trigger FlatPack to move the swapped out pages out of memory, releasing address space for accommodating new pages that will be compressed when data is written to memory.

FlatPack can be combined with other transparent and less transparent implementations to interact with the OS and release to it the free space created from compression.

3.9 Metadata

The described flexibility of placement requires supporting metadata to maintain page organization. Figure 7 summarizes the metadata used by FlatPack, which can be divided into three parts. The first part concerns page compaction. The MPA page location is 48 bits wide and indicates where in physical memory the compressed page resides. An additional 3 bits indicates the size of the physical page.

The second part of the metadata maintains block compaction. Any full-size slot can be in one of three states: *unused*, *used for compressed data*, or *used for lazily evicted data*. In addition, a slot may be occupied by one of three blocks (the native block N or one of the nearby blocks $N-1$ and $N-2$). By limiting each slot to one of three blocks, FlatPack can encode this information using three bits per slot. A minislot, similarly, may be either *unoccupied* or contain compressed data from one of three blocks. This requires two bits of metadata per minislot. The most complex page size (3.5kB) contains 48 full-size slots and 32 minislots. 16 full-size slots are reserved for their native blocks and always occupied, requiring no metadata. Consequently, each page requires at most $32 \times 3 + 32 \times 2 = 160$ bits of page compaction metadata.

The third and final part contains metadata for individual blocks. Each block has two bits associated with it, used to count failed compression attempts in order to reduce the frequency of retries. With 16 blocks in one page, this block metadata comprises 32 bits per page. In total, FlatPack requires a maximum of 243 bits of metadata per compressed page, roughly half of the 512 bits per page used by Compresso, LCP and MXT with 1KB blocks. However,

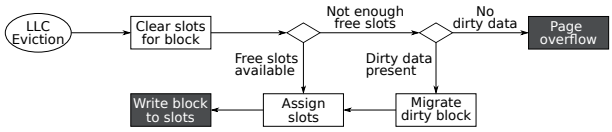


Figure 8: Overview of block writeback logic. Repacking of a block occurs on LLC eviction, when the block changes size.

MXT would require 8× more metadata than FlatPack to support the same fine granularity.

Metadata are managed on the page level, and cached on-chip in a Metadata Cache which has the same capacity as the system Translation Lookaside Buffer (TLB). Metadata are fetched from memory in parallel with TLB misses, which guarantees that block and page metadata are readily available by the time a request reaches LLC.

3.10 Block Migration

Figure 8 illustrates the logic flow of block writebacks. In the common case, sufficient slots and minislots are available in main memory, and the block can be written back directly. However, If a block cannot be packed successfully, slots must be freed up to accommodate it. The page metadata distinguishes compressed data from lazily evicted cache lines (i.e. uncompressed, dirty data). Blocks with lazily evicted cache lines consume more space in memory than necessary, to delay costly recompression. If such data prevents packing of a neighboring compressed block, the offending block is brought on-chip for recompression immediately, as shown in Figure 9. This way, physical space can be used for lazy evictions, and made available for actual compressed data on demand.

3.11 Page Migration

If no slots can be freed to pack an evicted block, the size of the compressed page is insufficient to contain its data. As a result, the data must be migrated to a larger page. The process of page growth is straight-forward. The page and block metadata are consulted to determine the smallest sufficient allocation size. Similarly to the initial allocation, an additional margin is added to the estimated size, to allow for flexibility to future growth. A new compressed page is set up and all blocks are read from physical memory. Finally, the compressed blocks are transferred to the newly allocated location.

Conversely, data updates during execution may cause blocks to shrink. In order to maximize the capacity gain of compaction, it is beneficial to detect and adapt to such changes. Each page's metadata is sufficient to identify pages whose data could fit within a smaller page allocation. When such a page is detected, FlatPack migrates it to a region for smaller pages to improve compaction.

3.12 Memory Interleaving

So far, the description of FlatPack has considered systems with one memory controller. However, address interleaving across multiple memory controllers can also be supported as follows. The minimum interleaving granularity is 4×MAG, keeping a single block access to a single channel. In addition, interleaving across n controllers requires FlatPack to consider groups of n pages together, rather than a single page as described above. Then, blocks within the group

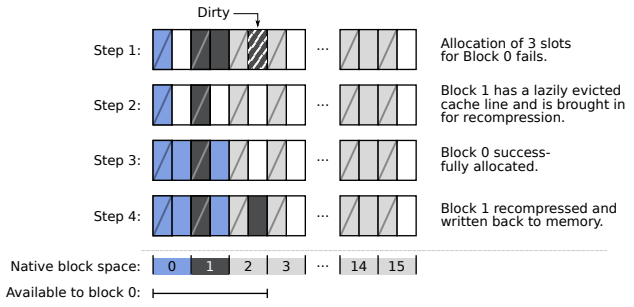


Figure 9: Block migration to support a growing block. Block 0 (3 MAGs) needs to be written back. In order to fit, block 1 is read from memory and recompressed to eliminate lazily evicted data.

of pages with the same (*block_id mod n*) are handled together as described in Section 3.4 as if they belonged to the same page, stored on the same channel. This requires metadata of all pages in the same group to be brought on chip together.

3.13 Hardware Overhead

Overall, as shown in Figure 6, FlatPack requires the following hardware changes: (i) modifications to the LLC, (ii) a Compressor [3], (iii) Metadata cache, and (iv) a FlatPack module before the memory controller. In particular, FlatPack employs a Decoupled-Sectored Cache, similar to other memory compression designs [10, 12], adding a 3.2% SRAM overhead due to the longer tag entries and back-pointer entries (18 bits per entry in total) with associated area and power costs. There is no latency overhead compared to a regular cache structure, because tag array and back-pointer array accesses are performed in parallel, as are the subsequent tag and suffix matches. Then, the logic for the final way-matching does not add additional delay. The compressor block is the major hardware addition, as in every memory compression design, causing the most significant area, power, and delay costs, reported in Table 2. Furthermore, a 64KB metadata cache is used as reported in Table 2. Finally, FlatPack adds a hardware module situated on the LLC side of the memory controller, which includes logic to issue memory requests to the memory controller and a small structure that keeps track of outstanding accesses and associated metadata arriving from the LLC. The area and power costs of this module are negligible. More importantly, it does not add latency to the memory access, although it is in the critical path. This is because the first fragment (MAG) of a compressed block is always at a fixed position and therefore the initial request can be sent immediately, bypassing the logic. In other words, no address calculation or other logic is required for the first memory access of a compressed block. Subsequent memory accesses to retrieve any subsequent fragments of the block are generated by the module and sent to the memory controller one per cycle according to the metadata.

Table 2: Simulation parameters.

Parameter	Configuration
Simulation Duration	1×10^9 instructions per core
CPU	O-o-O, 4-way issue @ 3.2GHz
L1 cache	64kB per core, 4-way, 1 cycle latency
L2 cache	256kB per core, 8-way, 8 cycle latency
L3 cache	1MB per core shared, 16-way, 15 cycle latency
Main Memory	4GB DDR4 and ¼-channel per core, 800MHz
Avail. Memory	50% of application footprint, at least 8× LLC
Page Fault Latency	8.6µs [38]
Metadata Cache	2048 entries, 64kB
Compressor	SC^2 lossless, 16-bit values
SC^2 comp/decomp leakage	33.6mW / 0.4mW
SC^2 comp/decomp dyn. energy	0.576nJ / 0.592nJ per operation
SC^2 VFT	7kB, 8-way, 16-bit values

3.14 Security Considerations

Compressed caches have been demonstrated to be susceptible to information disclosure via side channel attacks. SafeCracker outlines a set of circumstances where a secret value can leak from a process, to an attacker who is able to take measurements of the shared LLC [36]. By injecting known input data into a compressed victim cache line, the attacker is able to vary its compressibility and draw conclusions about the rest of the line's contents.

FlatPack presents a more complex target for this type of attack, since cache lines in LLC may be stored both compressed and uncompressed. An attacker must be able to manipulate the access pattern of the victim process to ensure that uncompressed copies of the secret are evicted and recompressed. In addition, the larger block size of FlatPack adds "noise" surrounding the secret data, reducing the certainty of an attacker's measurements.

4 EVALUATION

We evaluated FlatPack in an in-house simulator, implemented on top of Pin [23]. The simulator employs an interval-based processor model [14]. The memory hierarchy is modelled at cycle granularity, using DRAMSim2 for main memory [31]. McPAT [22] and CACTI [24] were used to model power and latency of the system considering 32nm technology. Operating frequency, latency and power consumption parameters for the SC^2 compressor are based on a place-and-route implementation for the same technology node [3]. These factors are used as configuration information for the simulations. The general properties of the simulated system are listed in Table 2.

Besides the baseline system, FlatPack (*F*) is further compared with (i) Compresso [6], labeled *C*, and (ii) LCP [28], labeled *L*. All three compressing designs use the same SC^2 compressor. The SC^2 compressor requires training data specific to each application to be effective. For each application, a profiling run is performed, where a randomized 10% subset of application data is gathered. This dataset is used to train the compressor for all designs, ensuring consistent compressor behavior. For evaluation we use the complete set of applications of SPECspeed 2017 [8], Graph500 [25], as well as ForestFire and PageRank from the SNAP suite [21].

Compression is applied to all non-code pages. This includes heap, stack, and data segments of the application itself, as well as those of shared libraries. All benchmarks use their respective *ref_speed* input data provided with SPEC 2017.

Table 3: Benchmarks and their active memory footprints.

Application	Footprint / core	Application	Footprint / core
bwav [8]	247MB	omnt [8]	16MB
cact [8]	521MB	perl [8]	20MB
cam4 [8]	59MB	pop2 [8]	73MB
deep [8]	688MB	roms [8]	133MB
exc2 [8]	16MB	wrf [8]	48MB
foton [8]	2320MB	x264 [8]	26MB
gcc [8]	33MB	xbmk [8]	16MB
imag [8]	34MB	xz [8]	525MB
lbm [8]	292MB	ffire [21]	11MB
leela [8]	16MB	gp500 [25]	255MB
mcf [8]	150MB	pgrank [21]	10MB
nab [8]	31MB		

A large benefit of memory compaction is the ability to avoid costly *page faults*, when data is swapped into memory from the slower nonvolatile storage. To investigate the effect of each design on page faults, the uncompressed baseline is profiled to determine its *active footprint* (the total number of unique pages in memory actually read or written) during the simulated phase of execution, as shown in Table 3. This baseline active footprint is used as a basis for page fault modelling, where the available physical memory is limited to 50% of the baseline footprint or at least 8× the LLC capacity (a minimum capacity limit of 32MB). This capacity limit is applied to all designs, including the baseline. Page faults are modeled as memory traffic and an additional delay of 8.6µs to account for OS handling and nonvolatile storage latency [38].

Each simulation is executed in the following steps: i) SC^2 compressor's Value Frequency table (VFT) is populated with data from the profiling run; ii) The application is run through its initialization phase; iii) the application's main phase is executed for one billion instructions per core, and statistics are gathered. The end of the initialization phase is manually selected such that statistics are collected during the application's primary processing phase.

In the following sections, we present our evaluation in two separate sets, a single-core system and a multi-core system where four instances of the same application are run on four processors.

4.1 Multi-Core Experimental Results

Performance benefits from memory compaction stem from the reduction of costly page faults as well as reduced memory traffic which leads to lower memory latency. Figure 10a shows the mean IPC achieved by each design across execution. FlatPack increases performance by 83%, with Compresso following at 34% and LCP offering a performance benefit of 22% over the baseline.

System energy consumption benefits from the performance increase as well as reduced memory activity. Figure 10b summarizes energy consumption for the processor core and memory hierarchy. FlatPack brings the total system energy down to 77%, while Compresso and LCP reach 87% and 92%, respectively.

The primary effect of memory compaction is a reduction in physical memory footprint. Figure 10c shows the achieved footprint for each design, by category. The *data* footprint consists only of the compressed data, excluding any compaction-related waste. Each design also introduces some *metadata* storage overhead, and a varying amount of *capacity waste*. *Granularity waste* stems from limited block granularity. *Shrink waste* is caused by a compressed block shrinking, leaving some of its original space unused. *Growth waste*

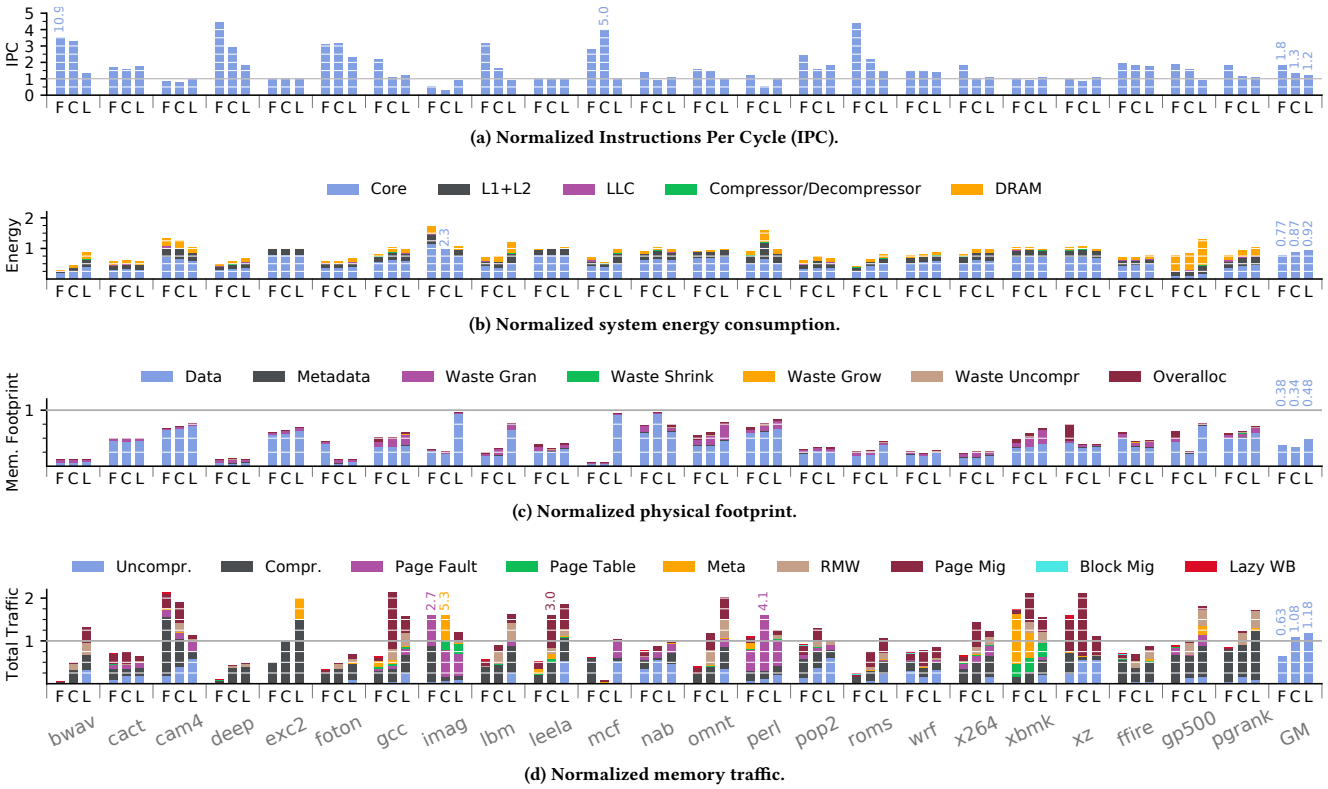


Figure 10: Multi-core results for FlatPack (F), Compresso (C) and LCP (L).

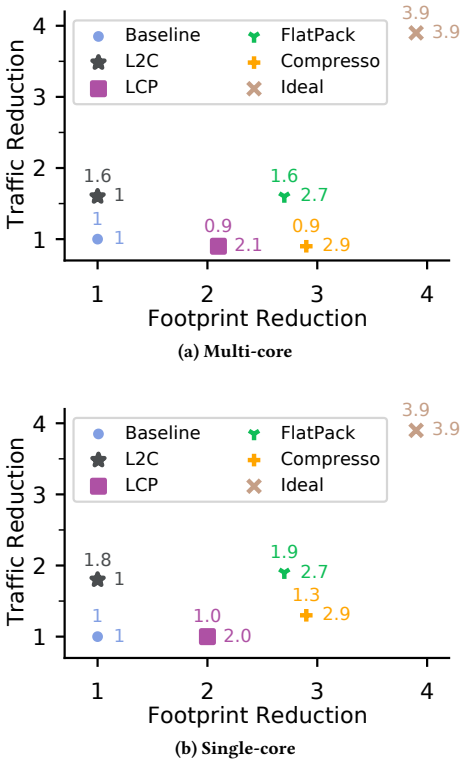
is caused by a compressed block growing to become an *exception*, leaving its old allocation entirely empty. *Uncompressed* waste represents blocks being left uncompressed for organizational reasons, even though their actual compressibility is better than 1:1. *Overallocation* is unused space due to the granularity of page allocation.

LCP's normalized footprint is 48% of the baseline and larger than competing designs mainly due to higher *granularity waste*. Compresso and FlatPack achieve similar totals, of 34% and 38%, respectively. FlatPack's flexibility introduces slightly higher compaction overheads than Compresso. This is evident in some benchmarks, e.g., *xz, gp500* and is attributed primarily to higher granularity waste and overallocation, as compressed pages appear to be smaller than initially predicted.

The raw compression ratio (average 3.9× across all applications) shows the average compressibility for the full memory footprint. This compression ratio sets the upper bound of compaction, and functions as an *ideal* reduction, as shown in Figure 11a. This ideal serves to give an estimate of the capacity overhead of each design, when compared to the achieved total footprint. The figure also illustrates a non-compacting design, L^2C [11], using the same compressor as described above. L^2C offers competitive traffic reduction, but does not affect memory footprint. Compared to the ideal 3.9× reduction, Compresso reaches 76%, FlatPack follows at 68% and LCP at 54%.

Memory compression also has the potential to reduce off-chip memory traffic, by transferring compressed blocks over the main memory bus. Like the footprint, the extent of this reduction is also bounded by the compression ratio. Many memory compaction schemes do not target traffic reduction, instead using additional traffic for the data movement required to support an adaptive compaction scheme. Figure 10d shows the volume of data transferred across the bus, broken down by cause. Various overheads are introduced by the evaluated compaction systems. *Metadata* traffic varies both due to the size of the metadata itself and its access patterns. *Read-Modify-Write* operations are necessary to update compressed data in memory at a finer granularity than the MAG, and take the form of additional reads from main memory. *Page Migration* is necessary to grow compressed pages upon overflow, or shrink them if possible. *Block Migration* traffic is induced by single compressed blocks being brought on-chip to update them with dirty data. *Lazy Writebacks* are used by FlatPack to delay block migrations. These overheads of memory compaction also prevent compaction systems from reaching a traffic reduction proportional to the ideal compression ratio. FlatPack page size estimation is successfully able to assign the correct initial size to 75% of pages. An additional 15% of pages are overestimated, preventing their first migration.

In total, FlatPack achieves a 1.6× reduction in memory traffic. This improvement is 20% compared to an ideal, theoretical traffic reduction roughly estimated by the average raw compressibility of



(a) Multi-core

(b) Single-core

Figure 11: Traffic and Footprint improvements. L^2C is a non-compacting memory compression system, here configured to use the same lossless SC^2 compression.

the memory footprints (3.9×). Compresso and LCP both add traffic overheads of 0.08× (4.2× ideal) and 0.13× (4.5× ideal), respectively. *Cam4* shows little spatial locality in its access pattern, illustrating the drawback of FlatPack's large compression blocks; while full compressed blocks are read from memory, parts of the data remain unused and become overhead. The same affects Compresso, to a lesser extent, as compressed overfetching does not serve as useful prefetching. A similar effect is seen in *imag*, where FlatPack's compressed traffic is increased by overfetching.

One significant traffic overhead from page compaction is *page migrations*. Normalized to the baseline's total memory traffic, FlatPack spends 4.0% on page migrations. The corresponding metric for LCP is 8.5% and for Compresso 16.3%. By reducing page migration overhead by 2×-4×, and eliminating a majority of RMW traffic, FlatPack offers a significant traffic benefit compared to the competing designs. FlatPack reduces memory traffic by 42% compared to Compresso, the next-best design. Finally, page faults make up a large portion of baseline memory traffic. The principal goal of reducing these page faults is achieved by all three designs. FlatPack reduces the mean number of page faults to 40%, while Compresso reaches 45% and LCP achieves a reduction down to 61%.

In summary, Compresso improves system performance and energy by 34% and 13%, respectively. FlatPack doubles these benefits, offering 83% better performance and 23% lower energy consumption

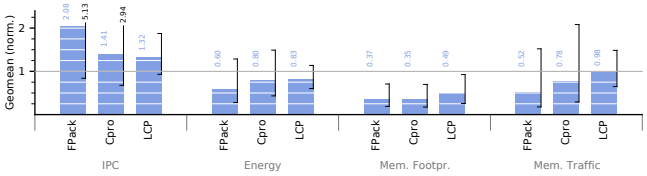


Figure 12: Single-core results for all designs. Geometric standard deviation in black.

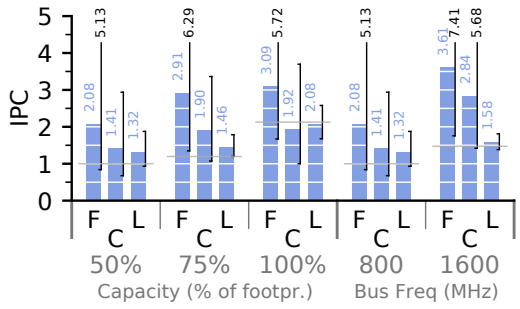
than a baseline system. FlatPack improves performance and energy consumption by 36% and 12%, respectively, versus Compresso.

4.2 Single-Core Experimental Results

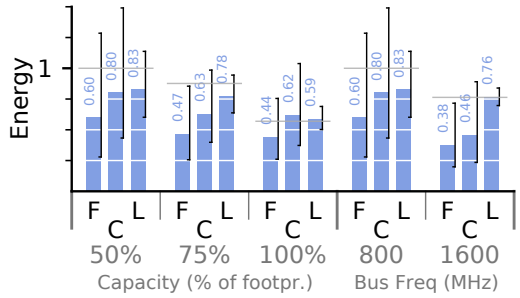
Figure 12 shows a summary of results from single-core experiments. Each metric is presented as the geometric mean across all applications, normalized to the baseline system. Although the evaluation of individual benchmarks is not depicted, most of them follow the same trends as in the multi-core experiments. Overall, system performance and energy, as well as memory footprint in single-core experiments improve similarly to the multi-core ones for all designs. On the other hand, memory traffic is slightly lower in single-core experiments of all design and is attributed mostly to less metadata traffic because the metadata cache size has not been scaled down.

Compresso yields a final memory footprint 35% of the baseline, closely followed by FlatPack at 37%. LCP brings footprint down to 49%. The average raw compressibility is compared to the achieved reduction in footprint and traffic in Figure 11b. Compresso achieves 65% of the ideal footprint reduction, FlatPack follows with 58% and LCP manages 34%. FlatPack reduces the mean memory traffic to 52% of the baseline. Compresso also reduces traffic, to 78% on average. LCP achieves an average 2% reduction. Similarly to the memory capacity, the raw compression ratio also indicates an ideal memory traffic reduction. FlatPack reduces memory traffic by 1.9×, which is 31% of the ideal. Compresso achieves a 1.3× reduction in traffic, 10% of the ideal. LCP does not affect memory traffic compared to the baseline. *Page migration* remains a significant overhead in the single-core system. Normalized to the baseline's total memory traffic, FlatPack spends 2.3% on page migrations. The corresponding metric for LCP is 4.9% and for Compresso 7.1%. LCP reduces the average total number of page faults to 66%, Compresso to 68% and FlatPack achieves a reduction to 72%. FlatPack increases system IPC by an average 108%, Compresso by 41% and LCP by 32%. The greatest performance boosts correlate to significant reductions in total memory traffic, indicating memory-bounded applications. FlatPack reduces average system energy to 60%. Compresso reaches 80% and LCP achieves an average of 83%.

In summary, the previous state-of-the-art Compresso improves system performance and energy by 59% and 20%, respectively. FlatPack exceeds these benefits, offering 108% better performance and 40% lower energy consumption than the baseline. FlatPack improves performance and energy consumption by 31% and 25%, respectively, compared to the second best design. Meanwhile, FlatPack maintains a memory capacity improvement within 6% of Compresso.



(a) Normalized system performance



(b) Normalized system energy

Figure 13: Means of system metrics for FlatPack (F), Compresso (C), and LCP (L) with varying memory capacity and bandwidth. All bars normalized to the 50% 800MHz baseline. Corresponding baselines are shown as horizontal lines. Geometric standard deviation in black.

4.3 Sensitivity to System Configuration

To further evaluate the impact of system parameters on FlatPack, we also present the result of two separate sensitivity analyses, shown in Figure 13. The first sweeps across available memory capacity, with limits set at 50%, 75% and 100% of the baseline footprint. The effect of this is a reduction in page faults, and thus an improvement in baseline performance. As the number of baseline page faults are reduced, the opportunity for memory compaction to improve performance also shrinks. *deep* sees a 13× IPC improvement from FlatPack at 50% memory capacity, which is reduced to 8× when capacity reaches 100%. While the benefits of compaction are reduced overall, the trend between the tested designs remains.

The second analysis illustrates the impact of memory bandwidth, comparing an 800MHz memory bus to a 1600MHz one. At the higher bandwidth, the performance benefits of compression are reduced. This is because compression primarily targets the bottleneck imposed by limited memory bus bandwidth. The effect is most notable in *bwav*, where a 14× IPC boost for FlatPack is reduced to 8× when memory bandwidth is doubled. As illustrated in Figure 13, the trend between the three compressing systems remains.

5 CONCLUSION

FlatPack is a novel approach to memory compaction, which allows compressed blocks to be packed fragmented within a page and share expansion space. It uses a hardware mechanism to dynamically reorganize pages when blocks are updated, without introducing any additional data movement. FlatPack offers memory capacity increases on par with state-of-the-art compaction systems, while improving on their memory bandwidth utilization by up to 67%. By leveraging compression and compaction to reduce data movement and costly page faults, FlatPack is shown to improve performance and energy consumption of a single-core system by 108% and 40%, respectively and in a multi-core system, the improvements are 83% and 23%, respectively. Compared to the best previous work, FlatPack improves performance by 31-46% and energy by 11-25%, while achieving a comparable memory capacity.

Acknowledgements

This work was supported by the Swedish Foundation for Strategic Research (contract number CHI19-0048) under the PRIDE project.

References

[1] Alaa R Alameldeen and David A Wood. 2004. Frequent pattern compression: A significance-based compression scheme for L2 caches. *Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep* 1500 (2004).

[2] A. Arelakis, F. Dahlgren, and P. Stenstrom. 2015. HyComp: a hybrid cache compression method for selection of data-type-specific compression methods. In *MICRO*. IEEE, Waikiki, Hawaii, USA, 38–49.

[3] Angelos Arelakis and Per Stenstrom. 2014. SC2: A statistical compression cache scheme. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, Minneapolis, Minnesota, USA, 145–156.

[4] Yanan Cao, Long Chen, and Zhao Zhang. 2015. Flexible memory: A novel main memory architecture with block-level memory compression. In *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, Boston, Massachusetts, USA, 285–294.

[5] Xi Chen, Lei Yang, Robert P Dick, Li Shang, and Haris Lekatsas. 2010. C-pack: A high-performance microprocessor cache compression algorithm. *IEEE transactions on very large scale integration (VLSI) systems* 18, 8 (2010), 1196–1208.

[6] E. Choukse, M. Erez, and A. R. Alameldeen. 2018. Compresso: Pragmatic Main Memory Compression. In *MICRO*. IEEE, Fukuoka, Japan, 546–558.

[7] Esha Choukse, Michael B. Sullivan, Mike O'Connor, Mattan Erez, Jeff Pool, David Nellans, and Stephen W. Keckler. 2020. Buddy Compression: Enabling Larger Memory for Deep Learning and HPC Workloads on GPUs. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. ACM/IEEE, Valencia, Spain, 926–939.

[8] Standard Performance Evaluation Corporation. 2017. SPEC CPU 2017. Retrieved 2021-07-30 from https://www.spec.org/cpu2017

[9] Magnus Ekman and Per Stenstrom. 2005. A robust main-memory compression scheme. In *SIGARCH C.A. News*, Vol. 33. ACM, New York, New York, USA, 74–85.

[10] Albin Eldstål-Damlin, Pedro Trancoso, and Ioannis Sourdis. 2019. AVR: Reducing Memory Traffic with Approximate Value Reconstruction. In *ICPP*. ACM, Kyoto, Japan, 1–10.

[11] Albin Eldstål-Ahrens, Angelos Arelakis, and Ioannis Sourdis. 2022. L2C: Combining Lossy and Lossless Compression on Memory and I/O. *ACM Trans. Embed. Comput. Syst.* 21, 1, Article 12 (jan 2022).

[12] Albin Eldstål-Ahrens and Ioannis Sourdis. 2020. MemSZ: Squeezing Memory Traffic with Lossy Compression. *ACM TACO* 17, 4, Article 40 (Nov. 2020), 40:1–40:25 pages.

[13] Peter A. Franaszek and Dan E. Poff. 2007. Management of Guest OS Memory Compression In Virtualized Systems. Patent US20080307188A1.

[14] D. Genbrugge, S. Eyerman, and L. Eeckhout. 2010. Interval simulation: Raising the level of abstraction in architectural simulation. In *HPCA*. IEEE, Bangalore, India, 1–12.

[15] E.G. Hallnor and S.K. Reinhardt. 2005. A unified compressed memory hierarchy. In *11th International Symposium on High-Performance Computer Architecture*. IEEE, San Fransisco, California, USA, 201–212.

[16] S. Hong, P. J. Nair, B. Abali, A. Buyuktosunoglu, K. Kim, and M. Healy. 2018. Attaché: Towards Ideal Memory Compression by Mitigating Metadata Bandwidth

Overheads. In *MICRO*. IEEE, Fukuoka, Japan, 326–338.
[17] Raghavendra Kanakagiri, Biswabandan Panda, and Madhu Mutyam. 2017. MBZip: Multiblock data compression. *TACO* 14, 4 (2017), 1–29.
[18] Jungrae Kim, Michael Sullivan, Esha Choukse, and Mattan Erez. 2016. Bit-plane compression: Transforming data for better compression in many-core architectures. In *ISCA*. ACM/IEEE, Seoul, Republic of Korea, 329–340.
[19] Sohan Lal, Jan Lucas, and Ben Juurlink. 2019. SLC: Memory access granularity aware selective lossy compression for GPUs. In *DATE*. IEEE, IEEE, Grenoble, France, 1184–1189.
[20] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W Keller. 2003. Energy management for commercial servers. *Computer* 36, 12 (2003), 39–48.
[21] Jure Leskovec and Rok Sosič. 2016. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1–20.
[22] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO*. IEEE, New York, New York, USA, 469–480.
[23] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, Vol. 40. ACM, New York, New York, USA, 190–200.
[24] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP lab.* 27 (2009), 22–31.
[25] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.
[26] David J. Palframan, Nam Sung Kim, and Mikko H. Lipasti. 2015. COP: To Compress and Protect Main Memory. In *42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. ACM, 682–693.
[27] Sungbo Park, Ingab Kang, Yaebin Moon, Jung Ho Ahn, and G. Edward Suh. 2021. BCD Deduplication: Effective Memory Compression Using Partial Cache-Line Deduplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. 52–64.
[28] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. 2016. Linearly compressed pages: a low-complexity, low-latency main memory compression framework. In *MICRO*. IEEE, Taipei, Taiwan, 172–184.
[29] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Michael A Kozuch, Phillip B Gibbons, and Todd C Mowry. 2012. Base-delta-immediate compression: Practical data compression for on-chip caches. In *PACT*. ACM, Minneapolis, Minnesota, USA, 377–388.
[30] Ashish Ranjan, Arnab Raha, Vijay Raghunathan, and Anand Raghunathan. 2020. Approximate Memory Compression. *IEEE TVLSI* 28, 4 (2020), 980–991.
[31] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A cycle accurate memory system simulator. *IEEE CAL* 10, 1 (2011), 16–19.
[32] Larry Seiler, Daqi Lin, and Cem Yuksel. 2020. Compacted CPU/GPU Data Compression via Modified Virtual Address Translation. *Proc. ACM Comput. Graph. Interact. Tech.* 3, 2, Article 19 (Aug. 2020), 18 pages.
[33] A. Seznec. 1994. Decoupled sectored caches: conciliating low tag implementation cost and low miss ratio. In *ISCA*. ACM/IEEE, Chicago, Illinois, USA, 384–393.
[34] Ali Shafiee, Meysam Taassori, Rajeev Balasubramonian, and Al Davis. 2014. MemZip: Exploring unconventional benefits from memory compression. In *HPCA*. IEEE, Orlando, Florida, USA, 638–649.
[35] R Brett Tremaine, Peter A Franaszek, John T Robinson, Charles O Schulz, T Basil Smith, Michael E Wazlowski, and P Maurice Bland. 2001. IBM memory expansion technology (MXT). *IBM Journal of Research and Development* 45, 2 (2001), 271–285.
[36] Po-An Tsai, Andres Sanchez, Christopher W Fletcher, and Daniel Sanchez. 2020. Safecracker: Leaking secrets through compressed caches. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1125–1140.
[37] Carl A. Waldspurger. 2003. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.* 36, SI (dec 2003), 181–194.
[38] Jisoo Yang and Julian Seymour. 2018. Pmbench: A micro-benchmark for profiling paging performance on a system with low-latency SSDs. In *Information Technology-New Generations*. Springer, New York, New York, USA, 627–633.
[39] Youtao Zhang and Rajiv Gupta. 2003. Enabling partial cache line prefetching through data compression. In *2003 International Conference on Parallel Processing, 2003. Proceedings.* IEEE, IEEE, Lyon, France, 277–285.
[40] Jishen Zhao, Sheng Li, Jichuan Chang, John L Byrne, Laura L Ramirez, Kevin Lim, Yuan Xie, and Paolo Faraboschi. 2015. Buri: Scaling big-memory computing with hardware-based memory expansion. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 3 (2015), 31.