MORC: A Manycore-Oriented Compressed Cache

Tri M. Nguyen
Princeton University
trin@princeton.edu

David Wentzlaff
Princeton University
wentzlaf@princeton.edu

ABSTRACT

Cache compression has largely focused on improving single-stream application performance. In contrast, this work proposes utilizing cache compression to improve application throughput for manycore processors while potentially harming single-stream performance. The growing interest in throughput-oriented manycore architectures and widening disparity between on-chip resources and off-chip bandwidth motivate re-evaluation of utilizing costly compression to conserve off-chip memory bandwidth. This work proposes MORC, a Manycore ORiented Compressed Cache architecture that compresses hundreds of cache lines together to maximize compression ratio. By looking across cache lines, MORC is able to achieve compression ratios beyond compression schemes which only compress within a single cache line. MORC utilizes a novel log-based cache organization which selects cache lines that are filled into the cache close in time as candidates to compress together. The proposed design not only compresses cache data, but also cache tags together to further save storage. Future manycore processors will likely have reduced cache sizes and less bandwidth per core than current multicore processors. We evaluate MORC on such future manycore processors utilizing the SPEC2006 benchmark suite. We find that MORC offers 37% more throughput than uncompressed caches and 17% more throughput than the next best cache compression scheme, while simultaneously reducing 17% of memory system energy compared to uncompressed caches.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: Parallel Architectures

Keywords

caches, compression, manycore

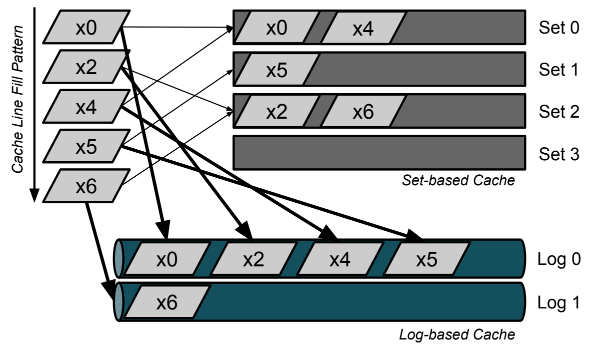


Figure 1: Cache line indexing is different between set-based and log-based caches. Log-based caches fill cache lines sequentially, regardless of addresses.

1. INTRODUCTION

Throughput-oriented computing is becoming increasingly important [1, 2]. Emerging examples of throughput computing occur in the enterprise (batch record processing, network processing), large data centers (bulk data serving, web-crawling, Map-Reduce applications, image and video transformation), scientific computing (Monte Carlo simulations), and at home (graphics processing in games, virus scanning). Generally, these workloads are latency tolerant, a characteristic often leveraged for better power efficiency. To sustain the growth of these applications, manycore architectures [3, 4, 5, 6, 7, 8, 9] prioritize energy efficiency and throughput over single-stream performance.

A major challenge with future manycore architectures is the need for additional off-chip memory bandwidth to feed the increasing number of threads and cores on a single chip. Although Moore's Law [10] is slowing, which makes including additional cores or computation on a single chip more difficult, the number of pins and the pin frequency that those pins can be toggled improve at an even slower rate, leaving a large and growing disparity in terms of bandwidth per core. This gap is known as the bandwidth-wall portion of the memory-wall [11]. Recent trends have begun to explore stacked DRAM architectures as a means to increase bandwidth. Unfortunately, stacked DRAM likely only provides a one-time increase in bandwidth and will be unable to keep pace with transistor scaling. Due to the

Operation	Energy	Scale
64b comparison (65nm) [12]	2pJ	1x
64b access 128KB SRAM (32nm) [13]	4pJ	2x
64b floating point op (45nm) [14]	45pJ	22.5x
64b transfer across 15mm on-chip [15]	375pJ	185x
64b transfer across main-board [16]	2.5nJ	1250x
64b access to DDR3 [17]	9.35nJ	4675x

Table 1: Energy of on-chip and off-chip operations on 64b of data. The energy consumption of off-chip data loads is almost a thousand times more expensive than on-chip access, which motivates localizing data on-chip as much as possible.

extreme bandwidth-starved nature of current and future throughput-oriented processors, any reduction in bandwidth can be applied directly to increasing throughput.

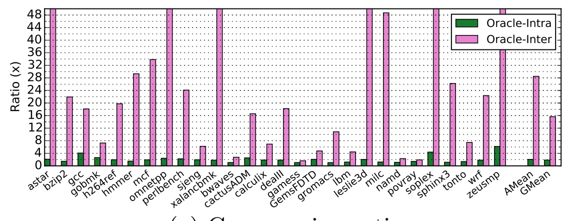
A secondary challenge that throughput-oriented processors need to tackle is the large energy cost associated with accessing off-chip memory. Table 1 shows the energy cost of accessing or operating on 64-bit words. Thousands of on-chip operations can occur for the cost of a single memory access. This fact strongly motivates techniques which can reduce off-chip memory accesses by using inexpensive on-chip computation.

In this work, we explore extreme cache compression as a means to increase throughput. To increase compression ratio, even at the expense of single-stream performance, we use expensive compression techniques along with inter-line compression. Prior work in cache compression [18, 19, 20, 21, 22, 23] has primarily focused on intra-line compression to minimize decompression latency. Even the recent work [24] which does compress across cache lines, is optimized for single-stream performance.

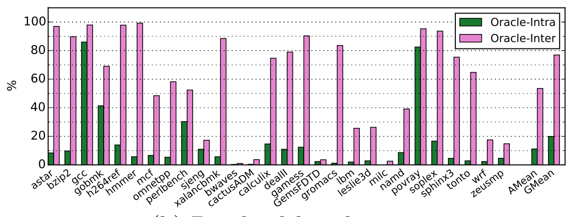
In contrast, this work explores cache compression solely to increase throughput, tolerating extremely long cache load latencies. We introduce MORC, a new log-based, Manycore ORiented, compressed Last-Level Cache (LLC) architecture that enables content-aware placement to compress similar cache lines together, previously impractical with commodity set-based caches. By utilizing logs, MORC compresses large numbers of cache lines together into a single active log, maximizing compression ratio while still providing a cache line look-up mechanism. The downside of higher compression ratio is that average access latency can increase due to long decompression time, especially when the required data is at the end of the log.

Figure 1 illustrates how data is filled into a log-based cache versus a set-based cache. In a log-based cache, data is located based on data commonality patterns, regardless of addresses; therefore, in Figure 1, addresses 4 and 5 which have similar data can be compressed together into the same log even though their addresses would indicate different cache indices. MORC not only uses logs to store compressed data, but also compresses tags together using base-delta compression. Through tag compression, MORC effectively overcomes tag overhead which can dominate storage in traditional compressed caches.

To further increase inter-line compression performance,



(a) Compression ratio



(b) Bandwidth reductions

Figure 2: Compression ratios and bandwidth reductions of ideal intra-line and inter-line compression

MORC utilizes a new compression algorithm, Large-Block Encoding (LBE), that dynamically chooses word granularity (i.e., 32, 64, 128, or 256-bits) for higher compression ratio. By compressing at large granularities, LBE significantly reduces compression pointer overhead.

MORC leverages an important insight to achieve compression ratios far beyond prior work: compressing larger blocks of data (e.g., at page-size) is likely to yield higher compression ratio than with smaller blocks (e.g., at cache block size). MORC takes this concept one step further and compresses similar cache lines regardless of physical address, even across memory pages. To show this potential, Figure 2 shows the compression ratio and bandwidth difference between an ideal *intra-line* (compression within a cache line) compression scheme, and an ideal *inter-line* (compression across cache lines) scheme. [1]. On average, intra-line compression achieves only 2x compression, reducing bandwidth usage by 20%, whereas inter-line achieves a staggering 24x compression ratio and almost 80% bandwidth reduction. In the bandwidth-exhausted regime, an 80% decrease in bandwidth usage can increase throughput up to 5 times. These results echo another study [25] on the limits of cache compression where caches can be ideally compressed up to 64x.

We evaluate MORC with the SPEC2006 benchmarks on a manycore processor with 128KB of cache per core and limited memory bandwidth per core which is likely representative of future manycore designs. We find that MORC's inter-line compression averages a 3x compression ratio, 50% more than the next-best design's 2x compression ratio. The increased effective cache size en-

[1]The study assumes set-based 128KB caches, where cache lines are compressed into 512-byte sets as much as possible, and evicted with LRU policy. Cache lines are compressed by splitting into 4-byte words and deduplicating them–within the cache line with intra-line, and across all cache lines with inter-line. Small values are further compressed by throwing away the most significant zeros (significance-based compression). Neither models have meta-data overheads (eg. pointers, tags, and fragmentation).

ables 17% more throughput than the next best scheme, at the same time MORC decreases memory system energy 17% over a baseline uncompressed cache. In the rest of this paper, we describe the MORC architecture in detail, compare MORC to three previous best-of-breed cache compression schemes, and explore design trade-offs in MORC.
This paper provides the following contributions:

- The design of a novel log-based, inter-line compression architecture which optimizes for throughput-oriented systems by favoring compression ratio over decompression latency.
- Introduction of tag-compression, enabling a larger range of maximum cache compression ratios.
- Introduction of a new data compression scheme, Large-Block Encoding (LBE), which provides high compression ratios.
- Introduction of content-aware compression by utilizing multi-log compression architecture.
- An evaluation of MORC versus best-of-breed prior work, measured in terms of off-chip bandwidth, throughput, and memory-subsystem energy.

2. BACKGROUND

2.1 Log-based cache

MORC utilizes *log-based* caches, in contrast to *set-based* caches prevalent in modern computers. Log-based caches are loosely based on the concept of log-based data structures widely used in software, where each new data item or modification is always appended to a *log*, preserving data integrity of previous entries. Appending data is one of the two operations allowed with logs; the other operation is flushing to reclaim cache space. Otherwise, logs and sets share many similarities, including that both data and tags are contained in an entry, tag checks can be done in parallel with data access, and that data is stored in conventional, indexible SRAMs.

Because logs do not support in-place modifications, they are often of limited use for uncompressed caches, as frequent write-backs can quickly saturate logs with outdated and invalid values. Even so, logs that contain only invalidated cache lines can be *reused* as if they are clean logs without needing to do a flush first.

2.2 Data compression in log-based caches

Preserving the data stream is necessary to enable modern stream-based compression algorithms. LZ [26] and gzip [27] compress data by using pointers to reference data which exists earlier in the uncompressed data stream (log). Thus, if cache writes were allowed to modify the stream (log) in-place, subsequent cache lines that reference the modified data will be corrupted and impossible to reconstruct. For the same reason, log-based compressed caches cannot have exact pointers to individual cache lines; decompression needs to start at the beginning of the stream (log) to build the correct data stream and dictionary. Unfortunately, this also means

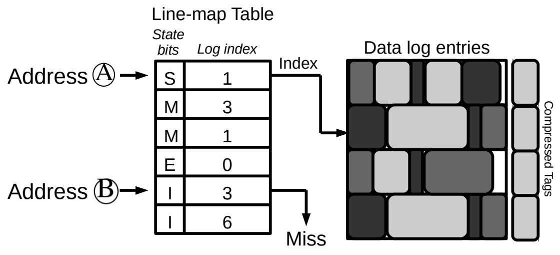


Figure 3: Two examples of a (direct-mapped) line-map table being accessed. Address Ⓐ is indexed to the first entry of the LMT, and then redirected to log #1. Address Ⓑ is similarly indexed and found to be invalid and not in the cache.

the average decompression latency would be longer for a log-based compressed cache than it would be in set-based caches. This is especially true when the required data is at the end of the stream.

Stream-based compression is natural and efficient for log-based caches. Compressing and appending a cache line simply requires that there is available space. Conversely, stream-based compression is complicated and costlier with set-based caches. For example, replacing an existing cache line likely means invalidating the subsequent compressed cache lines that reference the modified line, and/or re-compressing them.

While long access latency can harm single-stream performance, it can be a good trade-off in terms of energy (preventing expensive off-chip memory access) and throughput. Moreover, latency-hiding techniques, such as multi-threading in current manycore architectures, can be used to hide most of the extra access latency. Note that while these techniques can hide latencies, fundamentally they cannot solve the bandwidth-wall [11].

Additionally, log-based compressed caches sidestep two important problems with prior art: internal and external fragmentation. First, internal fragmentation is significant in set-based compressed caches, reducing the physical storage available to compress into. This is because small, fixed-size blocks of storage (*segments*) are used to support in-place modification. When compressed data is smaller than the allocated segments, the left-over storage cannot be used and can be up to 12.5% of storage for some designs [18]. Second, frequent defragmentation of segments is needed in a set-based compressed cache, unless the data-store is fully decoupled from the tag-store [19]. Defragmentation can be an energy-intensive and time consuming process with prior-work showing that it can increase LLC energy by almost 200%. [19]. Since log-based caches do not use segments, neither problem exists.

3. ARCHITECTURE

Instead of sets, MORC has fixed-size *logs* into which cache lines are compressed and *appended* incrementally. In-place modifications are not allowed and write-backs are always appended. Cache lines are filled into logs in temporal order, therefore, a line-map table (LMT) is

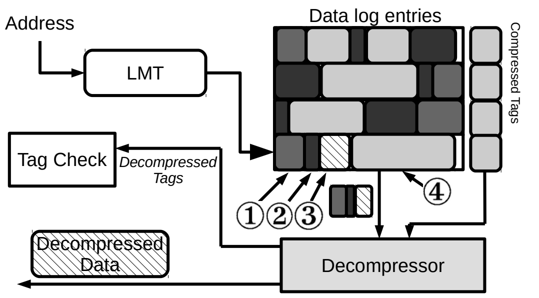


Figure 4: The decompression (cache access) process. The cache first checks the LMT. If valid, the compressed tags and data are streamed into the decompressor. The first two lines, ① and ②, are decompressed first before ③ can be decompressed. Decompression stops after line ③; line ④ is not processed.

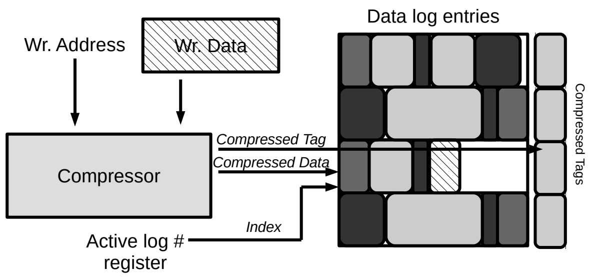


Figure 5: The compression (cache line fill) process. The writing address and data are fed into the compressor. The active log register holds the index of the log to be filled with data.

needed as an indirection layer to locate data by address. Cache addresses are indexed into the LMT, then redirected to the log that the data can be compressed into (for a fill) or can be found in (for a read/write request). In this respect, a MORC cache is similar to indirect caches [28, 20] except that the LMT is over-provisioned in order to track additional lines resulting from compression. Figure 3 shows a logical view of MORC with the LMT and log entries.

3.1 MORC Operations

To grasp the difference between MORC and set-based caches, we present a detailed description of read, fill, write-back, and eviction operations. We assume a non-inclusive cache design where write misses fill private caches first before eventually being evicted and written back to the LLC. Section 5.4.2 explains the rationale.

Read request (Figure 3) first checks the LMT. If the LMT entry is invalid, the cache line is guaranteed to not be in the cache, and the request is forwarded to memory. If valid, MORC still needs to decompress the tags and complete a tag-check (Figure 4) to ascertain hit or miss. We consider the case where the LMT entry is valid but tag-check indicates a cache-miss to be an "LMT aliased-miss." Once the tag-check verifies validity, MORC decompresses the indicated log only up to the needed data (known from tag location).

Fill request from memory is a result of a previous LLC miss. First, MORC allocates an LMT entry for the new cache line, and progresses to choose the best active log. The data is then compressed and appended to the log (as shown in Figure 5) and the LMT state bits and log index are set accordingly. If the allocated LMT entry is already valid before the fill, then an LMT-conflict occurs and an LMT entry eviction needs to be done first. When a log becomes full, MORC selects a victim log to flush and reclaims space.

Write-backs from private caches are also appended to logs similarly to fills from memory. In a non-inclusive cache, the LLC is not required to contain the cache line at write-back time, so an LMT entry is allocated if necessary. If it is already present, MORC changes the state to "Modified", and updates the LMT's log-index if the write-back is compressed into a different log.

Evictions have two forms in MORC: LMT-conflict evictions and a whole log eviction. LMT-conflict eviction happens when a cache fill is allocated to an already valid LMT entry. The tag entry for that cache line is marked invalid, and the data, if modified, needs to be decompressed and sent to memory.

A whole log eviction sequentially decompresses all entries in the log in order to reclaim space for a new active log. For each valid tag, MORC checks the state of the corresponding LMT entry: if modified, the decompressor decompresses data and writes it back to memory. In both cases, the LMT entry is marked invalid and available for future use. Note that a log flush only happens when a log is *evicted*; *reused* logs don't need to be flushed since all contained lines are invalid. Log evictions are infrequent and off of the critical path since the fill is forwarded to the processor core in parallel.

3.2 MORC Design Features

MORC has five distinct characteristics: (1) **log-based** storage efficiently enables dictionary-based stream compression algorithms, (2) **Line-Map Table** (LMT) enables flexible mapping of cache line to log, (3) multiple *active* logs enable **content-aware compression** for maximum compression performance, (4) **tag compression** reduces tag overheads, and (5) **Large-Block Encoding** (LBE) as a new compression algorithm optimizes inter-line compression.

3.2.1 Data storage

MORC's data storage is divided into multiple fixed-size log entries. Each log, typically limited to 512-bytes each to avoid high decompression latency, holds a variable number of cache lines, depending on the compressibility of the data in that particular log. Logs not only hold compressed data, but also allocate fixed storage for the compressed tags.

A MORC cache contains many logs, but only marks a subset as *active*; an active log is simply a log that is not full and can be appended to. Simple MORC implementations can have one active log, while more complex implementations can have multiple active logs, dynamically selected at compression time to maximize compression ratio. When a log is full, MORC closes it and chooses a new log. A victim log is chosen using any typical cache replacement policy, but this work studies FIFO for simplicity. Reusing closed logs with all cache lines invalidated (when the workload frequently writes back the same cache lines causing old copies to be invalidated) is given priority over choosing the next FIFO log. Figure 5 shows data being added to a MORC cache. Here, if the written data does not fit into the active log, the replacement policy selects a victim log to flush, reclaim, and compress new data into.

3.2.2 *Line Map Table*

The Line-map Table (LMT), shown in Figure 3 is an indirection layer that redirects memory requests to the respective (compressed) data and tag stores. The LMT is over-sized to track all of the cache lines at the maximum compression ratio.

As Figure 3 shows, an LMT entry contains two small fields: state bits and log index bits. The log index indicates which log the cache line has been compressed into. The entry *does not need to store the tag* but rather only the mentioned short log index, which points to the compressed tag-store where MORC decompresses and completes a tag-check. The state bits indicate whether the entry is invalid or valid, and if the cache line is modified. Checking for LMT entry validity substantially reduces average cache miss latency as MORC does not need to decompress the tags to resolve the miss.

Unlike conventional cache sets, logs and data in logs are not directly related to the LMT indexing. As a result, LMT index bits cannot be removed from the actual tag. Other indirect caches have the same needs [28]. Though Figure 3 shows a direct-mapped LMT, it can also be arranged to be set-associative. In a set-associative LMT, MORC needs to decompress and check tags in all logs that valid LMT entries point to. Experimental results show that a 2-way set-associative LMT reduces LMT-induced evictions from almost 20% to less than 5% compared to a fully-associated LMT. Hash-rehash or column-associated techniques [29, 30] can efficiently implement a 2-way LMT.

3.2.3 *Multi-log and Content-Aware Compression*

In order to improve compression performance, MORC can have multiple active logs to choose from when appending data. A higher compression ratio is achieved by choosing the most beneficial log to compress into. Intuitively, by having multiple logs instead of one, MORC increases the search space for data commonality, as opposed to a single log where data is strictly written sequentially. We loosely categorize this optimization as *content-aware compression.* Different compression algorithms could even be used for different data types which are stored in data-type specific logs. For instance, some logs can be dedicated to compress floating-point data using specialized compression algorithms [31].

To exhaustively extract performance out of multi-log compression, the inserting cache line is compressed to all active logs, but only the most fruitful log commits its dictionary state changes. Simpler heuristics could be used to save energy, but since compression energy is minuscule compared to, for example, MORC's decompression energy, we elected not to explore this direction.

One challenge with optimizing performance for multi-log is determining when to diversify distinct data across the active logs, as the policy of always choosing the best log could force the system to only use one log for most of the time. We found that a good diversifying algorithm is to insert a fudge factor, 5% for instance, to the scoring system such that when the best and the worst log compression sizes are within 5%, the cache line is seeded to the least-used log.

Code value	Distance (64B)	Precision bits
0-3	1-4	0
4-5	5-8	1
6-7	9-16	2
...	...	...
26-27	8,193-16,384	12
28-29	16,385-32,768	13
30-31	New base	0

Table 2: Distance coding for tags compression

3.2.4 *Tag Compression*

Tag overhead is a significant limitation of prior compressed caches. Over-allocating tags to support the maximum compression ratio can incur a significant overheads. For instance, assuming 40b tags, 8x allocation inflates the tag-store to 62.4% the size of the data-store, and for 16x to 124.8%. Clearly, extreme compression does not scale well with uncompressed tags.

To alleviate these overheads, we introduce tag compression. Because MORC appends cache lines in temporal order, tags are usually similar and hence highly compressible. We evaluated various compression algorithms, including LZ-based algorithms, and found that base-delta encoding (e.g., [21]) compresses tags best. In our implementation, tags are encoded as deltas to their immediate predecessor, using a scheme similar to DEFLATE's distance encoding [32], replicated in Table 2. Some modifications to the encoding include (a) one bit to indicate positive or negative deltas, (b) one bit to indicate validity, and (c) code 30-31 is used to indicate a new base when the difference between tags is larger than 2MB. One more base selection bit is added for the *multi-base* variation where two bases are tracked instead of just one to increase compression performance.

MORC is designed such that the tags and data can be accessed serially or in parallel. As base-delta compression is fast, and has been implemented to decompress 64Bytes in one cycle [21], we assume that the implemented tag compression can also decode up to 8 tags per cycle. As such, we have chosen in our results to access tags and then data sequentially to save energy.

3.2.5 *Large-Block Encoding*

Though MORC is flexible enough to be used with existing compression algorithms like C-Pack [23] or LZ, these compression engines are not optimal. With C-

Symbol	Description	Code	Symbol	Description	Code
u32	32b uncompressed	00	m64	64b match	1100
m32	32b match	01	z64	64b zeros	1101
z32	32b zeros	1010	m128	128b match	11100
u8	8b uncompressed	1011	z128	128b zeros	11101
u16	16b uncompressed	100	m256	256b match	11110
			z256	256b zeros	11111

Table 3: Compression prefixes for LBE

Pack, its constant pointer overhead (4-bits) per data word (32-bits) limits compression ratio to 8x. Increasing the dictionary size also increases the pointer overhead. For example, with a 512-byte dictionary the maximum compression ratio is reduced to 4.57x. In contrast, while LZ compresses general data well, its algorithm is hard to implement in hardware as commercial implementations can only encode and decode at 4-byte/cycle [33, 34].

We identify that the performance difference between LZ and C-Pack is that an LZ pointer can represent a variable and larger data block than C-Pack's 32-bit blocks. Motivated by this observation, we devise a new algorithm, called *Large-Block Encoding* (LBE), to efficiently compress multiple cache lines. Unlike LZ where arbitrary block length can be compressed to maximize compression, LBE compresses on aligned boundaries and limits compression to a fixed set of granularities: 32-bits, 64-bits, 128-bits, and 256-bits. Each of the different data sizes has its own logical dictionary, but only the 32-bit dictionary contains data, with the larger sizes pointing to entries in the 32-bit dictionary. Having this course-grain alignment restriction meshes well with data objects which also occur on coarse-grain alignment.

Compression algorithm: Logically, LBE reads input in 256-bit chunks. Compared to C-Pack which calculates eight hashes for the eight 32b chunks, LBE computes 7 additional hashes: four 64b, two 128b, and one 256b. LBE first finds matches in the 256b dictionary for the 256b hash. If not matched, it does the same for the 128b, then 64b, then 32b hashes. If there is a match in a dictionary at any point, LBE outputs the appropriate prefix (*m32, m64, m128, or m256*, as shown in Table 3) and a pointer to that entry. *z32* is preferred to *m32* for all zero data and has no associated pointer. Otherwise, LBE outputs *u32* (incompressible) followed by 32b of data, and makes a new 32b dictionary entry. Either *u8* or *u16* is used instead of *u32* when LBE can truncate the upper 24 or 16 zero bits.

Finally, before compressing the next 256b chunk, LBE allocates dictionary entries for any of the 64/128/256b chunks that failed to compress. Like a binary tree, each of these entries contains two pointers to dictionary entries one size smaller than itself. For instance, a 64b entry will point to two 32b entries, a 128b to two 64b entries, and a 256b to two 128b entries. During the decompression process, to decode an *m256* for example, LBE traverses down the binary tree to retrieve data stored in the 32b dictionary entries.

Implementation: We expect LBE to be nearly as simple and power efficient as C-Pack primarily because (a) its 32-bit dictionary is managed similarly (the dictionary is frozen when it is full), (b) there are only 7 additional hashes to compare for each 256b chunk (less than a factor of two), and (c) LBE encodes at the same 2 symbols/cycle rate. In fact, LBE is simpler because it does not try to compress individual 32-bits words besides simple upper zeros truncation. The addition of logical dictionaries is also likely to be energy efficient as matches are done using small hashes. LBE compares small hashes and then verifies that the data matches in the next pipe stage.

Scheme	Adaptive	Decoupled	SC2	MORC	MORCMerged
Tags	7.81%	0.00%	23.43%	7.81%	0.00%
Metadata	10.93%	8.59%	10.15%	17.18%	17.18%
Tags + Meta	18.74%	8.59%	33.58%	25.00%	17.18%
Comp. engine	.02mm^2	.02mm^2	NoData	.08mm^2	.08mm^2
Dict storage	128 Byte	128 Byte	18 KByte	1024 Byte	1024 Byte

Table 4: Overheads of compressions, normalized to cache capacity

3.2.6 *Tag-Data Storage Co-location*

An optimization to further reduce overheads is to completely remove the extra tag storage and co-locate tags with data. When more tag storage is needed, the extra tags overflow to the data log from the right. The behavior of data growing from the left and tags from the right is similar to how the heap grows from the bottom and the stack from the top.

We call this configuration "*MORCMerged*". The main drawback is that storage for compressed cache lines is reduced, leading to reduction in compression performance, although in cases where both tags and data have high compression ratio, co-locating both could produce a more efficient use of storage. Section 5.4.5 in the results shows that merging only decreases compression ratio slightly.

3.3 Overhead Analysis

Like prior compression schemes, there are three overheads associated with MORC: tag over-provisioning, compression metadata (the LMT), and compression engines. Assuming a 128KB cache, a 48-bit physical address space, 16-way sets (for prior-work), and 512B logs (for MORC), Table 4 compares tags and metadata overheads. The table shows MORC with enough LMT entries to hold 8x compressed data. The last column shows MORCMerged, described in Section 3.2.6, with the same LMT overheads but with no extra tags. Overall, the area overheads of MORCMerged is 17.18%, much less than the next-best prior work (SC2).

The area and energy overhead of compression engines can be a non-trivial design point. Because LBE is based on C-Pack, we can estimate its area based on the synthesis results presented by Chen [23]. In the experiments, we sized LBE's dictionary to be 512-bytes. Estimated in 32nm, C-Pack compressor and decompressor are both .01mm^2 in size. We conservatively increase the size eightfold to .08mm^2 . For reference, a 16-way 256KB cache in 32nm is 2.12mm^2 [13].

Another metric to measure area overhead is dictionary size. C-Pack in Adaptive and Decoupled requires

Core	2.0GHz, in-order x86 1 CPI for non-memory instructions
L1 Caches	32KB per-core, private, single-cycle latency 64B block size, 4-way set associative
LLC Caches	128KB per-core, shared non-inclusive, 14-cycle latency 64B block size, 8-way set associative
Memory system	FCFS memory controller, closed-page DDR3 1600MHz, 9-9-9 sub-timings
Decompression throughput (C-Pack/SC2/LBE)	8B/8B/16B per cycle

Table 5: System configuration

M0	h264ref_2, soplex, hmmer_1, bzip2, gcc_8, sjeng, perlbench_2, hmmer, sphinx3, zeusmp, gobmk_2, perlbench_1, h264ref, dealII, gcc_5, sjeng
M1	gobmk_2, gcc_2, astar_1, h264ref_2, gobmk_1, h264ref_1, bzip2_1, gcc_1, gobmk_4, bzip2_5, h264ref_2, gcc_4, xalancbmk, astar_1, bzip2_5, bzip2_5
M2	bzip2_2, perlbench, astar_1, perlbench, bzip2_5, sjeng, omnetpp, gcc_1, bzip2, h264ref, gcc, gobmk_4, perlbench_1, omnetpp, omnetpp, gcc_7
M3	hmmer_1, sjeng, bzip2_2, mcf, gcc_5, bzip2_5, hmmer, gcc_1, perlbench_1, gcc_4, hmmer_1, astar_1, astar, astar, gcc_5, h264ref
S0	bwaves x 16
S1	bzip2 x 16
S2	gcc x 16
S3	h264ref x 16
S4	hmmer x 16
S5	perlbench x 16
S6	sjeng x 16
S7	soplex x 16

Table 6: Multi-program workloads. Each has 16 programs, randomly chosen

64-bytes for compression and decompression each, for a total of 128-bytes. SC2 needs 18KB to support the Huffman compression flow [24]. As MORC allocates 512-bytes for each compression and decompression engine, the basic single-log implementation needs 1024-bytes of dictionary storage.

Naive multi-log implementations with 8 active logs contain 8 compression engines and 1 decompression engine, thus increasing the area overheads to .72mm^2 and dictionary size to 4608B, but we assume that time-division multiplexing can be applied to share one compression engine with multiple active logs, reducing the requirements to .08mm^2 and 1024B respectively.

4. METHODOLOGY

We evaluate our design using the SPEC2006 benchmark suite and PriME [35] simulator. PriME was augmented to capture data values in the caches to measure compression performance. The default system configuration is shown in Table 5. SPEC2006 benchmarks are run with as many reference inputs as possible; the additional inputs are indicated by an underscore and number.

We run a set of publicly available [36] profiled representative regions [37] for shorter simulations and higher accuracy. Single-program workloads (Section 5.1) use the 130M instruction trace set (100M warm-up, 30M actual instructions), while multi-program (Section 5.2) use 1 billion instruction regions where applications run the full 1B instructions but are only profiled for the first 250M instructions. These multi-program workloads are evaluated with two sets of workloads described in Table 6: (a) 4 sets of "mixed" applications, randomly chosen, and (b) 8 sets of replicated, "same" application.

Table 7 shows the energy model for a 32nm process; access energy numbers are per cache line. For SRAM and caches we use the 32nm models from CACTI [13]

L1 static power	7.0 mW	LLC static power	20.0 mW
L1 access energy	61.0 pJ	LLC data energy	32.0 pJ
C-Pack compression energy	50.0 pJ	LBE compression energy	200 pJ
C-Pack decompression energy	37.5 pJ	LBE decompression energy	150 pJ
SC2 compression energy	144 pJ	DRAM static power per core	10.9 mW
SC2 decompression energy	148 pJ	64B access off-chip energy	74.8 nJ

Table 7: Energy simulation parameters

ITRS-LOP. For DRAM energy, we use Micron's power calculator [17], assuming a quad-channel, 4-rank per channel, 9 chips per rank DIMM. For LBE, we scale up C-Pack's energy to take into account the increased dictionary size from 64-bytes to 512-bytes and 16B/cycle.

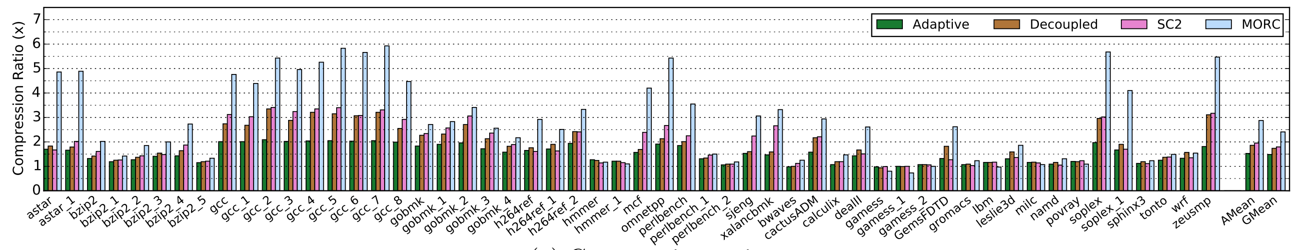
Compression ratios are sampled every 10M instructions as the ratio of valid cache lines over uncompressed cache capacity. Bandwidth usage is miss per kilo instruction (MPKI) scaled to 1B instructions. As IPC is mainly a metric of single-thread performance and not of throughput workloads where some latency to memory can be tolerated, we also estimate throughput using a four-thread, coarse-grain, multi-threading model. In this model, one thread is executed continuously until an L1 cache miss, at which point the next thread is swapped in. If the cache miss is serviced before the thread is swapped in again, the cache miss latency is considered hidden and there is no loss in throughput. Contrarily, if the cache miss needs to be serviced from memory which may take thousands of cycles, and other threads are also taking cache misses, then the core is forced to stall. We estimate the latency-tolerance of the workloads by measuring the average number of cycles between L1 misses, then subtract it from the compressed LLC access latency to calculate the core's non-stalling throughput. This latency is workload-dependent: lower for memory-intensive benchmarks and highest for compute-bound workloads.

In addition to uncompressed caches, we also compare to Adaptive [18], Decoupled [19], and SC2 [24] compressed caches. These schemes are evaluated with perfect LRU replacement policy. For fairness, both Adaptive and Decoupled were evaluated with C-Pack even though the original Adaptive design used FPC [38]. Baseline LLC load latency is 14 cycles; Adaptive, Decoupled, and SC2 all add 4 extra cycles for decompression; and MORC's variable decompression latency is 16 output bytes per cycle. Compression energies are estimated as in Table 7.

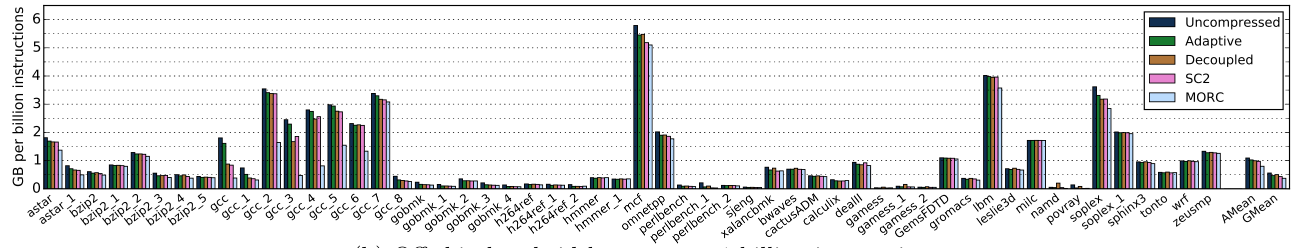
The evaluated MORC is allocated with 2x storage for tag-store, and enough LMT entries for a maximum compression ratio of 8x. The LMT is arranged in column-associated style [29] to emulate 2-way associativity. By default, MORC uses 512-byte logs, LBE for compression algorithm, 8 active logs for multi-log compression, and tags compression with 2 bases.

5. RESULTS

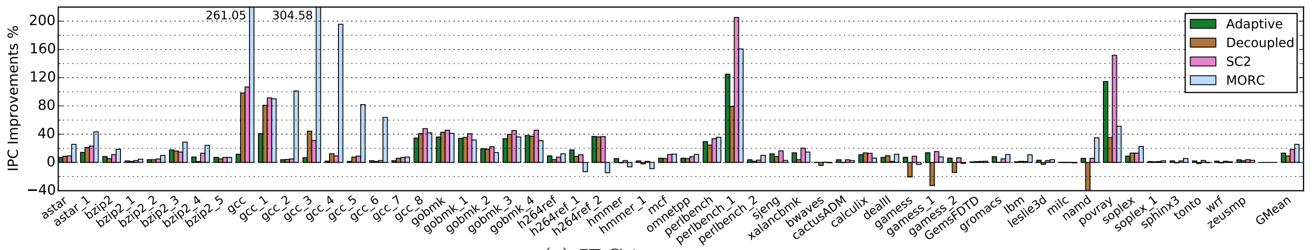
Our evaluation has two parts. In the first part, we evaluate MORC and other compression schemes on the basis of compressibility, bandwidth savings, IPC, and throughput improvements. We report these results for single-program in Section 5.1, and multi-program in



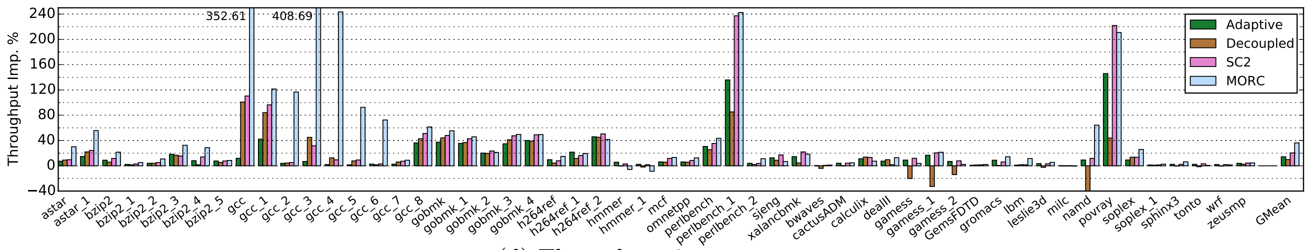
(a) Compression ratio



(b) Off-chip bandwidth usage per 1 billion instructions



(c) IPC improvements



(d) Throughput improvements

Figure 6: Single-program simulations. Each program is statically allocated 100MB/s of bandwidth.

Section 5.2. The second part studies the energy efficiency of inter-line compression (Section 5.3), and explores MORC's design space (Section 5.4).

The results show that MORC, by virtue of inter-line compression, is much better at data compression than prior art. As a result, MORC suffers less cache misses, saves more bandwidth, and extracts more throughput from less off-chip communication. Energy-wise, MORC consumes less than other schemes because of fewer accesses to memory, which outweighs the additional decompression energy from inter-line compression.

5.1 Single-program results

5.1.1 Compression performance

MORC outperforms prior art in compression ratio. In Figure 6a, the average effective cache sizes of MORC is 2.9x, with some workloads compressed close to 6x (*astar*, *gcc*, *omnet*, *soplex*, *zeusmp*) and many more close to or beyond 3x. Compared to the averages of Adaptive, Decoupled, and SC2 at 1.5x, 1.8x, 1.9x respectively, MORC consistently compresses the workload better.

Scrutinizing the compression results, we make two observations. First, in workloads with abundant zeros like *zeusmp* and *gcc* which should be compressible even with intra-line techniques, prior art runs out of tags. Both Adaptive (max 2x) and Decoupled/SC2 (both max 4x) run into this issue. In contrast, MORC perceives almost no tag restrictions thanks to tags compression.

Second, we observe that there exists an abundant amount of non-zero data duplication across cache lines, of which MORC exploits to achieve a big leap in compression performance. To support this observation, we profile and plot the usage distribution of LBE's encoding symbols in Figure 7. To simplify the chart, the *mX* portion of the left bars include both actual *mX* and *zX* symbols. Notably for some workloads, non-zero *m256* usage is significant; some examples include *cactusADM*, *gamess*, *leslie3d*, and *povray*, where the left *m256* bar is distinctively higher than the right *m256* bar. By

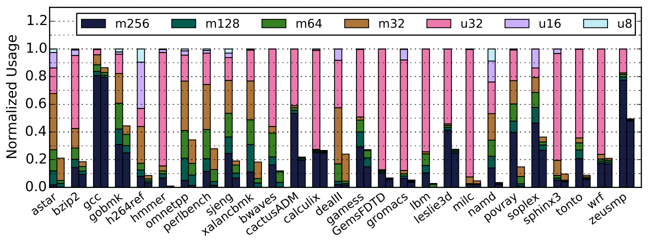


Figure 7: Normalized LBE encodings distribution; encoding symbols are explained in Table 3. The left columns show the total usage, while the right ones depict the portions that are all zeros. Distribution is weighted to data-size each encoding represents.

contrast, workloads like *gcc* are composed mostly of zeros. In both cases, MORC achieves much better compression ratios as an *m256* symbol represents 8 times the amount of data as an *m32* for the same space usage. Other symbols like *m64* and *m128* are also useful for workloads where data duplications occur at smaller granularities, like *mcf*, *omnet*, and *perlbench*. As Section 3.2.5 argues, big symbols like *m64*, *m128*, *m256* are needed to achieve high data compression ratio. Lastly, a subset of benchmarks, especially *h264ref*, benefits from significance-based compression (*u8*/*u16* where upper zero bits are truncated).

5.1.2 Bandwidth & Throughput

Figure 6b shows the bandwidth usage of the evaluated compressed caches. Note that the bandwidth savings originate solely from larger effective cache sizes; none of the schemes compresses the memory channel. By average-means, MORC reduces off-chip bandwidth by 27.0%, while the next best, SC2, only by 10.8%. While not always true, higher effective cache sizes strongly correlate with higher bandwidth reductions. Some programs–like *astar*, *gcc*, and *perlbench*–exhibit more pronounced impacts than others–like *dealII* and *zeusmp*. The differences in working set size among different benchmarks explain this behavior: those that fit after compression use much less bandwidth than those that do not.

This is prominent for FP workloads which have huge working sets. For example, a characterization study of SPEC2006 [39] shows that the LLC miss-rate of *cactusADM* is the same for caches between 128KB and 2MB in size. Since the baseline cache is 128KB in size, cache compression ratios from 1x to 15x will all have the same miss-rate to memory.

Figure 6c shows the IPC improvements resulting from bandwidth reduction under a 100MB/s per-core bandwidth cap. On average, MORC achieves a 22% IPC gain over the baseline, edging out SC2's 20% even when MORC is not optimized for single-stream performance. When applying multi-threading to cache compression to hide load latency of the LLC, throughput differences between MORC and other schemes increase even further, to 37% for MORC and 20% for SC2 as seen with Figure 6d. Multi-threading's effectiveness with MORC ranges from no improvements (eg. *bzip2*, *mcf*, *omnet*), to moderate (eg. *astar_1*, from 42% to 49%), to significant (eg. *povray*, from less than 60% to over 200%). Multi-threading benefits MORC more than other schemes because those are already optimized for IPC and thus cannot take advantage of latency-hiding techniques.

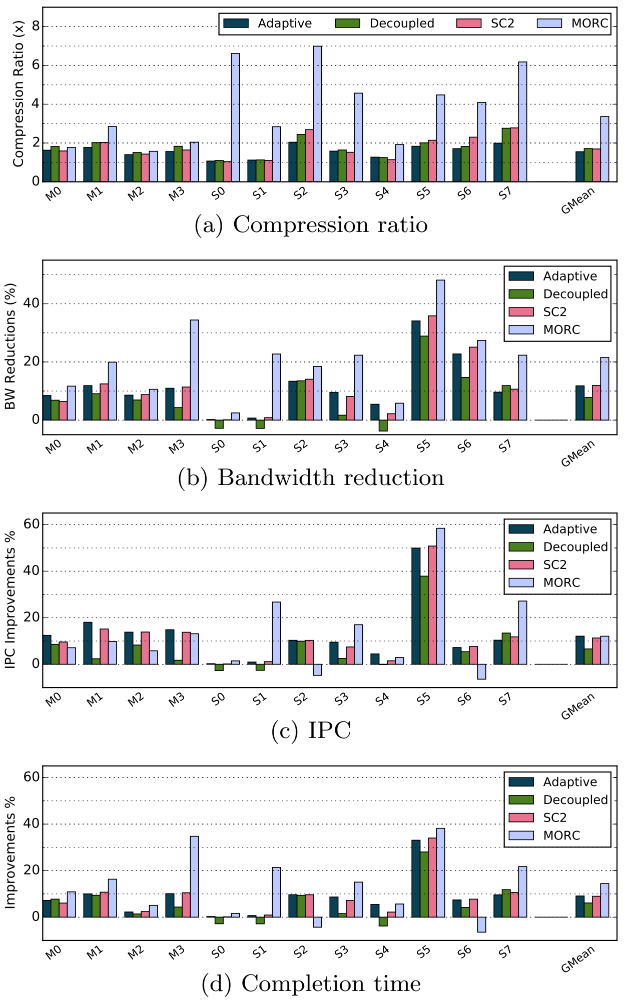


Figure 8: Multi-program simulations. The *Mx* workloads are mixed programs, and *Sx* are similar, as listed in Table 6. All workloads have 16 threads in total.

5.2 Multi-program

We simulate multi-program workloads (Table 6) to understand the behavior of a shared LLC when executing a mix of programs. In these simulations, the system has 1600MB/s of sharable bandwidth in total (100MB/s per thread with 16 threads). Consistent with single-program results, Figure 8a shows that MORC has good compression ratio compared to the baselines.

With almost 4x on average and up to 7x compression ratio, MORC is far superior to the next-best scheme which sees only 1.75x on average. Since MORC extracts commonality across applications, compression performance for the same-benchmark workloads (*Sx*) is generally great. The *Sx* workloads represent systems where like workloads are co-scheduled and grouped to the same machines. However, as evidenced in some workloads (*S4*, and to some extend *S1*), slight asynchronism in execution phases between threads applies more stress to the compression engines and severely decreases MORC's compression performance. Techniques that identify and synchronize threads at the instruction level [40] can completely eliminate threads asynchronism and greatly increase compression performance.

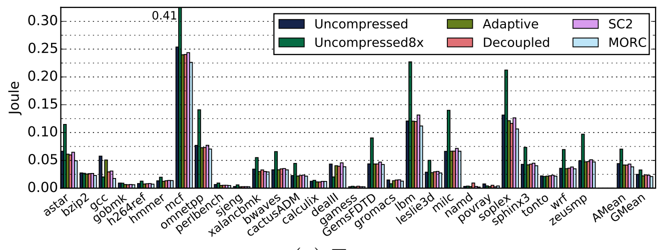
MORC, as well as other compression schemes, performs less well with random mixes of workloads (*Mx* set). With MORC, compressing data streams from different applications together to the same log (or pool of logs with multi-log) is not as effective as compressing the streams separately. The same situation applies to SC2 where the fix-sized centralized dictionary has to be shared among multiple programs and thus is less effective. Figure 8a shows neither Decoupled nor SC2 achieving compression not much better than Adaptive.

Consequentially, in Figure 8b, off-chip memory bandwidth is greatly reduced with MORC: by 20% on average and up to almost 50%. The magnitude of savings is still subjected to the workloads' working-set sizes, thus workloads with high compression ratios can have little savings (*S0*), while those with low compression ratios can achieve large savings (*M3*). Generally though, MORC compresses multi-program workloads better than prior work and thus achieves better bandwidth savings.

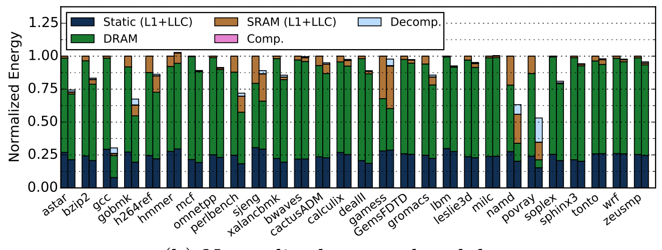
IPC, in Figures 8c, and completion time, in Figure 8d, measure different aspects of throughput. The IPC plot measures the geometric mean of IPC across all 16 applications, generally suited to compare the trend of speedups for latency-critical applications. In contrast, the completion time plot measures the run time of the longest running application. This approximates a user using a Hadoop cluster where phases' run time is limited by the tail latency. Short completion time is also significant to performance in GPGPU where the longest running threads limit performance of compute kernels.

Overall, MORC performs superior to prior-work with the *Sx* sets and about the same for *Mx* sets. Measuring IPC, MORC can obtain a performance increase of almost 60% (*S5*), and between 17% to 27% performance increase for workloads in which prior work achieves significantly lower (*S1*, *S3*, and *S7*). Especially with the *Mx* workloads, MORC attains lower-than-expected IPC gains from bandwidth reductions since bandwidth is shared and bandwidth-intensive phases of one application can be overlapped with compute-intensive phases of another application, effectively using bandwidth more efficiently than in single-program simulations. Nevertheless, that just means total system bandwidth is not fully utilized, and throughput can be further increased by running more threads.

Completion time speedups show a larger advantage for MORC, mainly in *Mx* workloads where mixed workloads have their tail latency reduced significantly. First, since Figure 8c plots the unweighted means, and CPU-bound applications are negatively affected by longer LLC latency, the positive speedups from memory-intensive workloads are concealed in the unweighted IPCs. The completion time plot in Figure 8d, however, accentuates performance gain of the most memory-intensive workloads, and shows MORC completing *M3* 35% faster, up from 17% for IPC improvement.



(a) Energy



(b) Normalized energy breakdowns

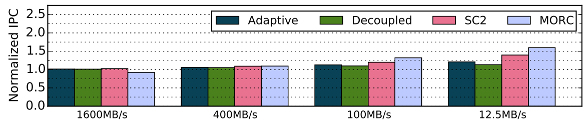
Figure 9: Memory subsystem energy. In the breakdowns chart, left columns compare the baseline to MORC's energy usage in the right columns.

5.3 Memory Energy Efficiency

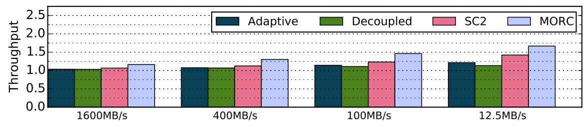
DRAM accesses represent a large portion of both total running time and energy, even after cache compression. Prior work has shown that compressed caches are more efficient than larger uncompressed caches thanks to significantly lower static and dynamic energy power [24]. Figure 9a compares the memory subsystem energy, including compression engine (but not CPU core energy) of cache compression schemes (at 128KB) and baselines (at 128KB and 1MB). On average, MORC reduces 17.0% of memory-system energy, more than the other compression schemes and is accomplished by removing more DRAM accesses. Moreover, compared to the 1MB baseline, MORC is more energy efficient due to lower static power of a smaller SRAM.

Decompression energy is small relative to DRAM access, L1 data array, or LLC energy. Hence, when MORC reduces the number of DRAM accesses, memory system energy goes down proportionately. Figure 9b normalizes MORC to the baseline with energy broken down; in most cases, the reduction in DRAM accesses (and in some cases static energy due to shorter execution length) outweighs the added energy from compressions.

Observe that compression is efficient with log-based caches since subsequent compressions are incremental and no cache line compaction is needed. Decompression energy is more substantial in MORC because it needs to decompress from the beginning of the compression stream. Particularly, decompression power is more significant with *namd* and *povray*, but is also compensated by substantial reductions in DRAM power. Other workloads (*gcc*, *astar*, *gobmk*) exhibit noteworthy energy savings, often up to 68%.



(a) IPC



(b) Throughput

Figure 10: Normalized performance at other bandwidth availabilities per thread.

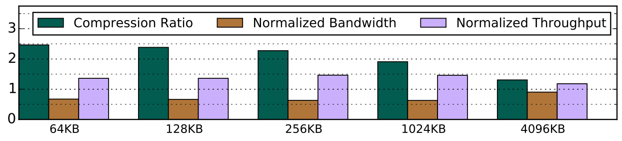


Figure 11: MORC's performance at other cache sizes.

5.4 Sensitivity Studies

5.4.1 Bandwidth and Cache Size

The single and multi-program results show that for a future design point (1024 cores with 128KB LLC per core and eight memory channels), MORC can achieve big performance gains. For completeness, we evaluate MORC at other bandwidth availabilities. As expected, MORC's single-stream performance (Figure 10a) suffers when there is abundant bandwidth–MORC slows the system by almost 7% at 1600MB/s. However, with multi-threading (Figure 10b), there is no throughput loss; and at extreme bandwidth starvation (12.5MB/s, a possible design point for 2020 [41]) MORC improves throughput by 63% . In Figure 11, MORC's bandwidth savings and throughput gains are shown to be strong consistently for different cache sizes from 64KB to 1MB. In this range, bandwidth savings are between 33% and 37%, which translate to throughput improvements from 35% to 46%. Only at 4MB and beyond that most workloads fit in-cache and cache compression is no longer beneficial.

5.4.2 Measuring write-back effects on logs

Frequent writes to the LLC can adversely affect log-based caches as writes append and old space is not reclaimed until the log is flushed. Figure 12 shows that a simple optimization of *not* inserting the fetched line on first write (*non-inclusive*) significantly reduces the presence of invalid old data over the *inclusive* behavior. In the non-inclusive model, only subsequent write-backs are written to the data-store. Additionally, write-back data for most workloads is highly compressible, comparable to compression ratios of valid data in Figure 6a. Furthermore, reusing logs reduces the frequency that a valid log has to be flushed. Therefore, we were not compelled to devise a fall-back strategy for overwritten data in MORC.

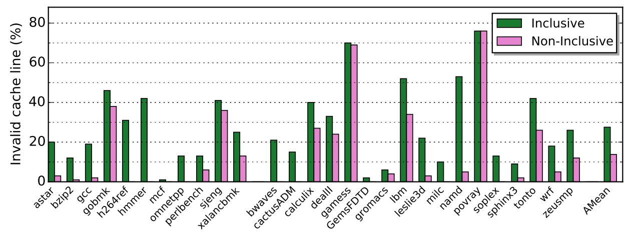
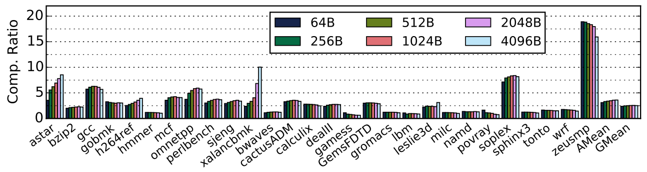
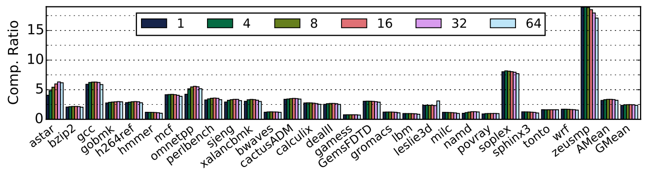


Figure 12: Write-back-induced invalid ratio in MORC. Compression is disabled to accentuate invalidations.



(a) Varying log sizes (in bytes); with 8 multi-logs.



(b) Varying number of active logs; with 512-byte logs.

Figure 13: MORC's compression performance across various log sizes and multiple active logs using LBE

5.4.3 Log-size and number of logs

Intuitively, larger log sizes (to store longer compression streams) and more active logs (to sort more data patterns) should enable even greater compression performance for MORC. On the contrary, with the limit studies in Figure 13, assuming unlimited tags and LMT entries, we observe that the configuration of 512-byte logs with 8 active logs (out of total of 512 in 128KB caches) is almost optimal.

5.4.4 MORC latency distribution

Figure 14 explores the variability of decompression latency with MORC. The workloads exhibit fairly even distribution of accesses to both data in the front (lower latency) and in the back of the log (longer latency). In other words, the usefulness of a cache line in the logs is position-independent.

5.4.5 Shared data/tag logs

Figure 15 shows the evaluation of MORCMerged, described in Section 3.2.6, against the default configuration. Overall, MORCMerged sacrifices minimal compression performance, less than 0.5x for most workloads. *omnetpp* suffers the biggest drop in performance because its memory access pattern is not as compressible as, for example, *soplex* or *zeusmp*. Interestingly, even with reduced total storage, co-locating tags and data can use space more efficient and achieve higher compression ratios, as evidenced with workloads like *hmmer* and *lbm*.

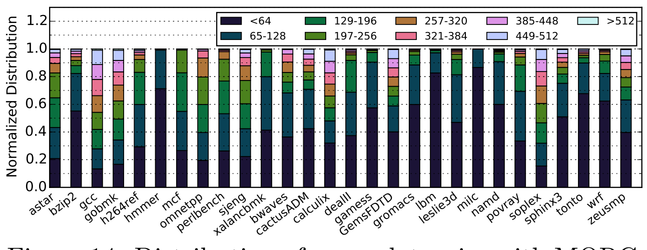


Figure 14: Distribution of access latencies with MORC, assuming 16B/cycle output speed.

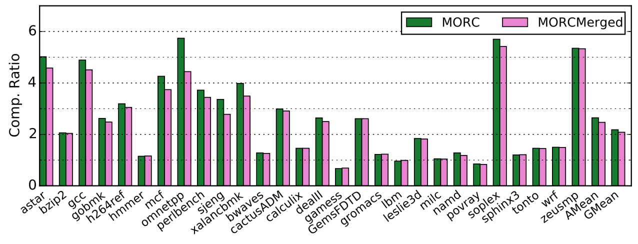


Figure 15: Compression performance trade-off between baselines (separated tag/data-stores) and merged where extra tags overflow to and co-locate with data-store.

6. RELATED WORK

Most prior work on LLC compression compresses single cache lines [18, 23, 19, 20, 22]. While one recent scheme [24] compresses inter-line, it still prioritizes single-stream performance over throughput. In contrast, MORC is optimized for throughput. MORC was evaluated against three best-of-breed cache compression schemes: Adaptive [18], Decoupled [19], and SC2 [24].

Adaptive [18] uses a basic cache organization with sets and ways. Each set over-provisions tags by 2x for a maximum compression of 2x, and allocates storage for data in segments of 8-bytes. To minimize metadata, segments for a given compressed line are continuous, which necessitates defragmentation when write-backs expand and consume more segments. Decoupled [19] aims to reduce tag-overheads, increase maximum compression, and eliminate defragmentation by using super-tags and pointers to individual segments. Skewed Compressed Cache [42] has similar performance to Decoupled, but is designed to be easier to implement [42]. Finally, SC2 [24] is most similar to MORC because it maintains a system-wide dictionary which can compress data across cache lines. However, as the shared dictionary is limited in size, the amount of data that SC2 can track is limited to the most common words. SC2's architecture, and consequently overheads, is similar to Adaptive's; SC2 cannot scale past 4x maximum compression without incurring expensive tag-overheads. In the evaluation, we observe that while SC2 is better than the other prior-work, it performs worse than MORC due to limited tags and limited frequent data in the dictionary. Lastly, SC2 needs software procedures to adapt the dictionary to new data over time; MORC requires no software.

Related work to LBE include C-Pack's [23], FPC [38], and LZ (software [27, 26], hardware [33, 34, 43]). LBE is loosely based on C-Pack, but is optimized for inter-line compression. FPC [38], strictly an intra-line compression algorithm, performs similarly to C-Pack, in our own evaluations and prior studies [23]. LZ and its derivatives, are well-known and frequently used in modern software to compress general data. In our (not-shown) studies, we found that LZ, as a direct replacement to LBE, has similar compression performance.

Memory link compression [44, 45] is complementary to cache compression. MORC does not compress the link and reduces bandwidth demands solely through higher effective cache sizes.

7. CONCLUSION

This work presents MORC, a cache compression scheme that aggressively compresses hundreds of cache lines together to maximize throughput in future bandwidth-starved manycore architectures. Its contributions include a log-based compressed cache architecture, a tag compression technique to lower the tags overheads, a novel data compression algorithm, and a content-aware compression technique. In future systems where throughput and energy efficiency is of greater emphasis than single-stream performance, MORC's bandwidth savings deliver 37% throughput improvement and 17% reduction in memory system energy.

8. ACKNOWLEDGEMENTS

This work was partially supported by the NSF under Grant No. CCF-1217553, AFOSR under Grant No. FA9550-14-1-0148, and DARPA under Grants No. N66001-14-1-4040 and HR0011-13-2-0005. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

9. REFERENCES

[1] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, *et al.*, "Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 451–460, ACM, 2010.

[2] Y.-K. Chen, J. Chhugani, P. Dubey, *et al.*, "Convergence of recognition, mining, and synthesis workloads and its implications," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 790–807, 2008.

[3] J. Jeffers and J. Reinders, *Intel Xeon Phi coprocessor high performance programming*. Newnes, 2013.

[4] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, *et al.*, "Larrabee: a many-core x86 architecture for visual computing," *ACM Transactions on Graphics (TOG)*, vol. 27, no. 3, p. 18, 2008.

[5] S. R. Vangal *et al.*, "An 80-tile sub-100-w teraflops processor in 65-nm cmos," *Solid-State Circuits, IEEE Journal of*, vol. 43, no. 1, pp. 29–41, 2008.

[6] J. S. Kim, M. B. Taylor, J. Miller, and D. Wentzlaff, "Energy characterization of a tiled architecture processor with on-chip networks," in *Proceedings of international symposium on Low power electronics and design*, pp. 424–427, ACM, 2003.

[7] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, "On-chip interconnection architecture of the Tile Processor," *IEEE Micro*, vol. 27, pp. 15–31, Sept. 2007.

[8] S. Bell *et al.*, "Tile64 - processor: A 64-core soc with mesh interconnect," in *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pp. 88–598, Feb 2008.

[9] C. Ramey, "Tile-gx100 manycore processor: Acceleration interfaces and architecture," in *Proceedings of Hot Chips Symposium*, 2011.

[10] G. E. Moore, "Cramming More Components Onto Integrated Circuits," *Electronics*, Apr. 1965.

[11] D. Burger, J. R. Goodman, and A. Kägi, *Memory bandwidth limitations of future microprocessors*, vol. 24. ACM, 1996.

[12] P.-J. Chuang, M. Sachdev, and V. Gaudet, "A 167-ps 2.34-mW Single-Cycle 64-Bit Binary Tree Comparator With Constant-Delay Logic in 65-nm CMOS," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 61, pp. 160–171, Jan 2014.

[13] S. Thoziyoor, N. Muralimanohar, and N. P. Jouppi, "CACTI 5.0," *HP Laboratories, Technical Report*, 2007.

[14] S. Galal and M. Horowitz, "Energy-efficient floating-point unit design," *Computers, IEEE Transactions on*, vol. 60, no. 7, pp. 913–922, 2011.

[15] R. Ho, K. W. Mai, and M. A. Horowitz, "The future of wires," *Proceedings of the IEEE*, vol. 89, no. 4, pp. 490–504, 2001.

[16] TECHNICK.NET, "PCB Impedance Calculator," *http://www.technick.net/public/code/cp_dpage.php?aiocp_dp=util_pcb_imp_microstrip*.

[17] Micron Technology, "DDR3 System-Power Calculator," *www.micron.com/support/power-calc*.

[18] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *Proceedings of International Symposium on Computer Architecture*, pp. 212–223, IEEE, 2004.

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

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

[21] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: practical data compression for on-chip caches," in *Proceedings of international conference on Parallel architectures and compilation techniques*, pp. 377–388, ACM, 2012.

[22] S. Kim, J. Lee, J. Kim, and S. Hong, "Residue cache: a low-energy low-area L2 cache architecture via compression and partial hits," in *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, pp. 420–429, ACM, 2011.

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

[24] A. Arelakis and P. Stenstrom, "SC2: A statistical compression cache scheme," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pp. 145–156, IEEE, 2014.

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

[26] I. Pavlov, "LZMA SDK," *www.7-zip.org/sdk.html*, 2007.

[27] L. P. Deutsch, "GZIP file format specification version 4.3," 1996.

[28] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design," in *Proceedings of International Symposium on Computer Architecture*, pp. 107–116, IEEE, 2000.

[29] A. Agarwal and S. Pudar, "Column-associative Caches: A Technique For Reducing The Miss Rate Of Direct-mapped Caches," in *Proceedings of International Symposium on Computer Architecture*, pp. 179–190, IEEE, 1993.

[30] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache performance of operating system and multiprogramming workloads," *ACM Transactions on Computer Systems (TOCS)*, vol. 6, no. 4, pp. 393–431, 1988.

[31] M. Burtscher and P. Ratanaworabhan, "FPC: A high-speed compressor for double-precision floating-point data," *Computers, IEEE Transactions on*, vol. 58, no. 1, pp. 18–31, 2009.

[32] L. P. Deutsch, "DEFLATE compressed data format specification version 1.3," 1996.

[33] AHA, "AHA Data Compression," *http://www.aha.com/data-compression/*.

[34] Indra, "Indra Products," *http://www.indranetworks.com/products.html*.

[35] Y. Fu and D. Wentzlaff, "PriME: A parallel and distributed simulator for thousand-core chips," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pp. 116–125, IEEE, 2014.

[36] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 52:1–52:12, Nov. 2011.

[37] H. Patil and T. E. Carlson, "Pinballs: Portable and Shareable User-level Checkpoints for Reproducible Analysis and Simulation," in *Proceedings of the Workshop on Reproducible Research Methodologies (REPRODUCE)*, 2014.

[38] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for L2 caches," *Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep*, vol. 1500, 2004.

[39] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation," *Web Copy: http://www.glue.umd.edu/ajaleel/workload*, 2010.

[40] M. Mckeown, J. Balkind, and D. Wentzlaff, "Execution Drafting: Energy Efficiency Through Computation Deduplication," in *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, pp. 432–444, IEEE Computer Society, 2014.

[41] O. Villa, D. R. Johnson, M. O'Connor, *et al.*, "Scaling the power wall: a path to exascale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 830–841, IEEE Press, 2014.

[42] S. Sardashti, A. Seznec, D. Wood, *et al.*, "Skewed Compressed Caches," in *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, pp. 331–342, IEEE, 2014.

[43] R. B. Tremaine *et al.*, "IBM memory expansion technology (MXT)," *IBM Journal of Research and Development*, vol. 45, no. 2, pp. 271–285, 2001.

[44] M. Thuresson, L. Spracklen, and P. Stenstrom, "Memory-link compression schemes: A value locality perspective," *Computers, IEEE Transactions on*, vol. 57, no. 7, pp. 916–927, 2008.

[45] V. Sathish, M. J. Schulte, and N. S. Kim, "Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads," in *Proceedings of international conference on Parallel architectures and compilation techniques*, pp. 325–334, ACM, 2012.