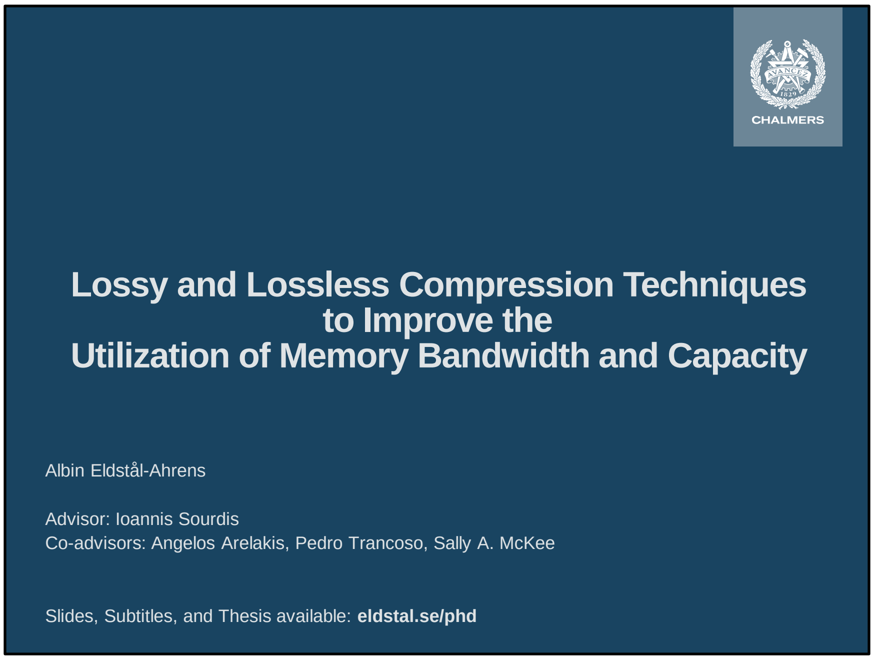


Lossy and Lossless Compression Techniques to Improve the Utilization of Memory Bandwidth and Capacity

Albin Eldstål-Ahrens

Advisor: Ioannis Sourdis
Co-advisors: Angelos Arelakis, Pedro Trancoso, Sally A. McKee

Slides, Subtitles, and Thesis available: **eldstal.se/phd**

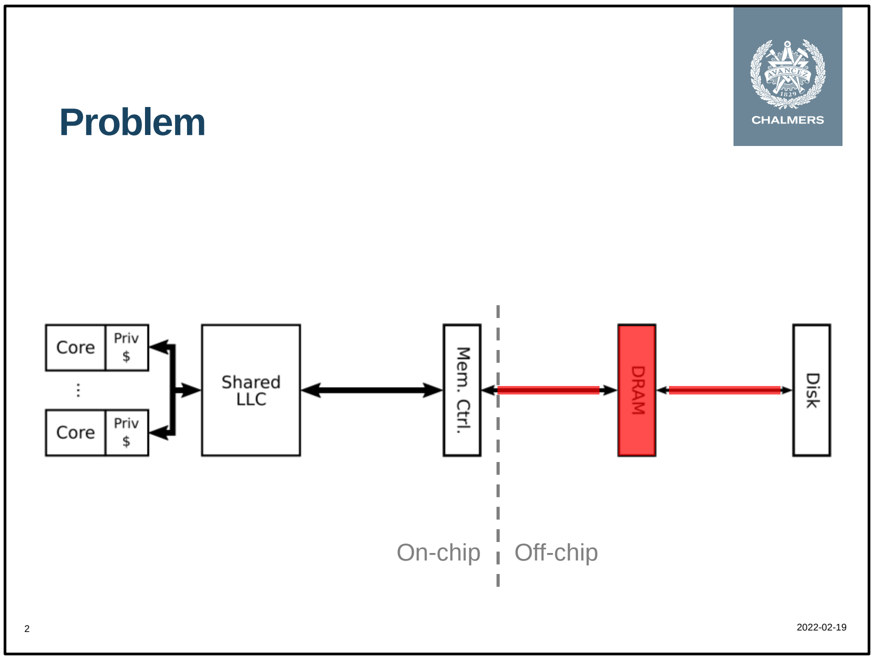
Hello everyone, thank you for coming!

Before we begin, I'd like to say that if you're following along at home, this presentation, its subtitles, and the thesis itself are available online at this address: eldstal.se/phd

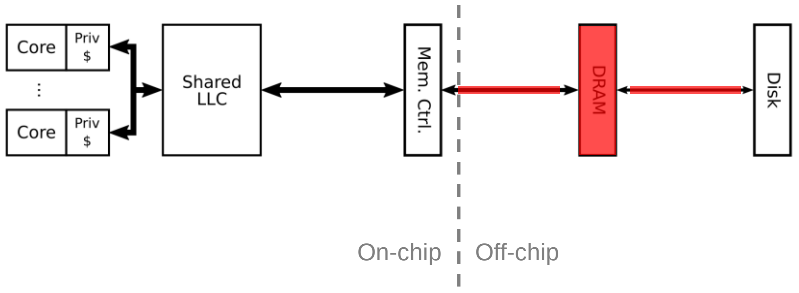
My name is Albin Eldstål-Ahrens, I'm a Ph.D. Candidate at Chalmers University of Technology, in beautiful Gothenburg, Sweden.

I'm here to present my doctoral thesis, titled **Lossy and Lossless Compression Techniques to Improve the Utilization of Memory Bandwidth and Capacity**.

My advisor throughout has been Yiannis Sourdis, and I've been co-advised by Angelos Arelakis, Pedro Trancoso and Sally McKee.



Problem

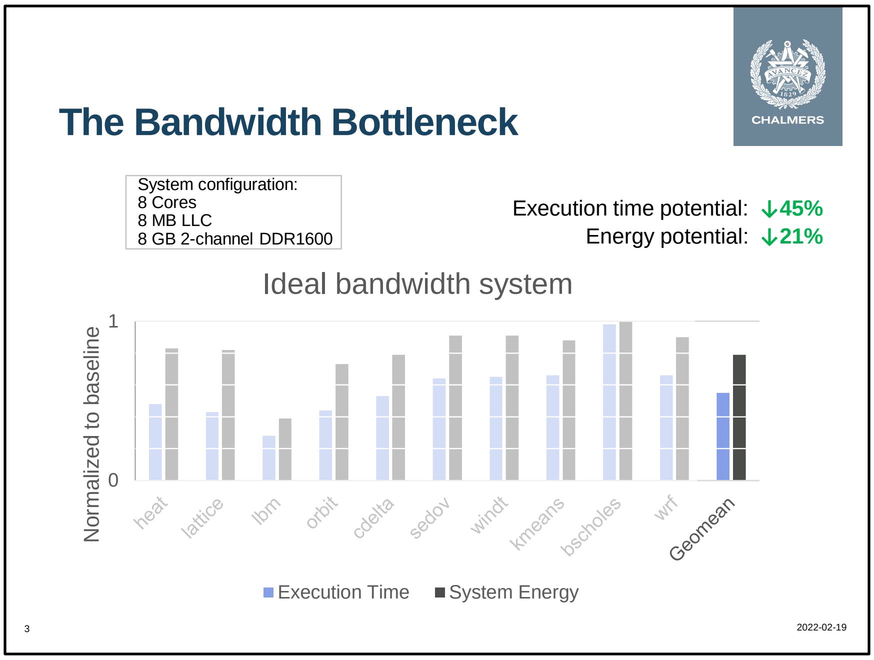


2 2022-02-19

In the common memory hierarchy, there is a stark difference in performance between on-chip elements (caches) and off-chip elements (main memory and persistent storage).

One cause of this is the limited bandwidth of the main memory bus, which can become a bottleneck in memory-intensive applications.

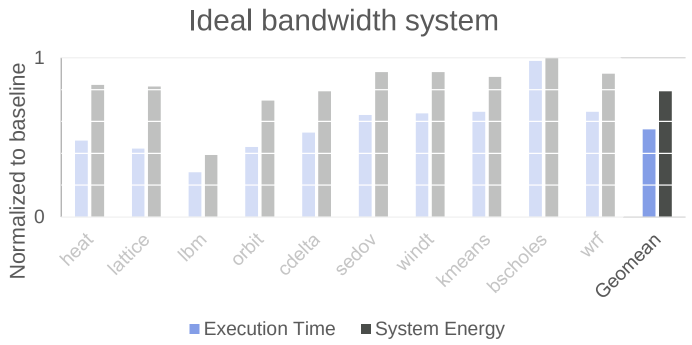
Another bottleneck is the capacity of main memory. When main memory is exhausted some data must be swapped between memory and disk, a slow and energy-intensive operation known as a *page fault*.



The Bandwidth Bottleneck

System configuration:
8 Cores
8 MB LLC
8 GB 2-channel DDR1600

Execution time potential: ↓45%
Energy potential: ↓21%

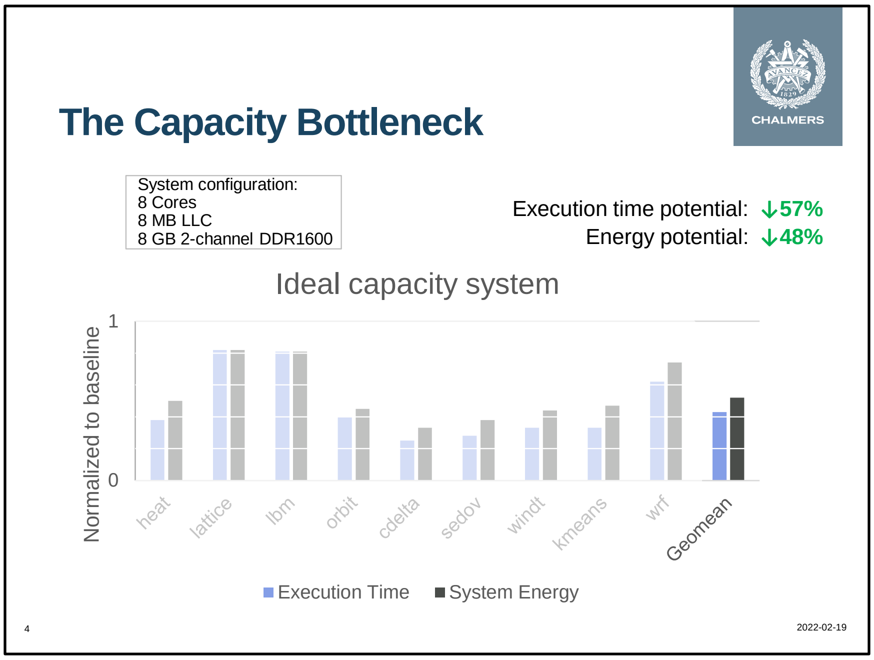


3 2022-02-19

This figure shows the potential benefits, if the bandwidth bottleneck could be eliminated.

1.0 in this figure corresponds to a baseline system, with a realistic memory bus.
The bars show metrics for a theoretical system with an infinite bus bandwidth, and lower is better.

As you can see, performance can be improved by an average of 45% for these benchmarks.
System energy consumption can be reduced by 21%.

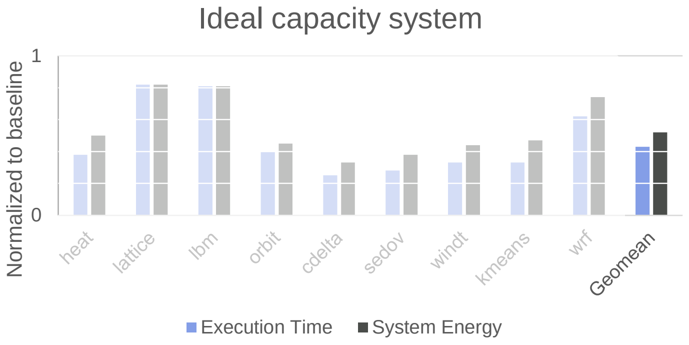


The Capacity Bottleneck

CHALMERS

System configuration:
8 Cores
8 MB LLC
8 GB 2-channel DDR1600

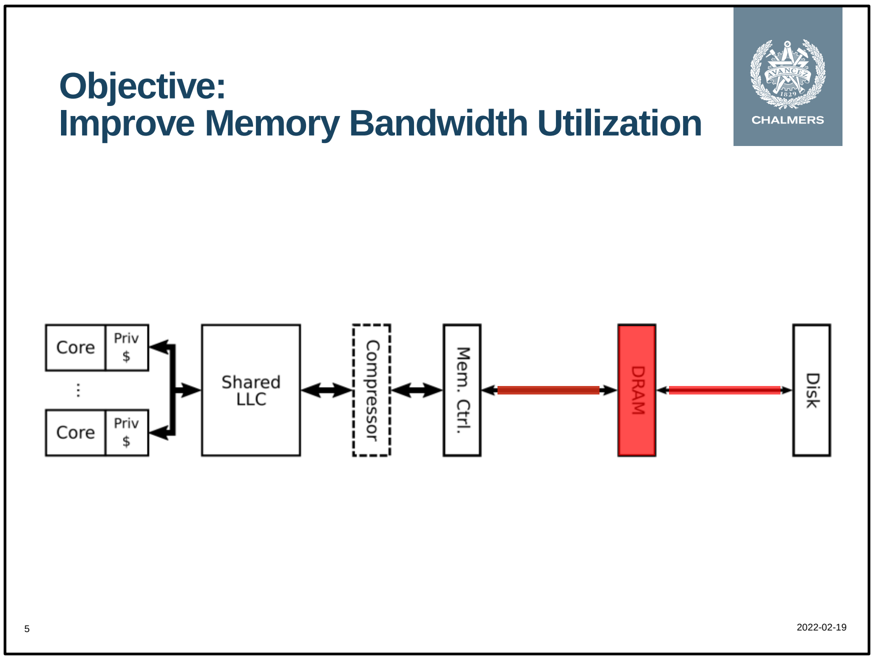
Execution time potential: ↓57%
Energy potential: ↓48%



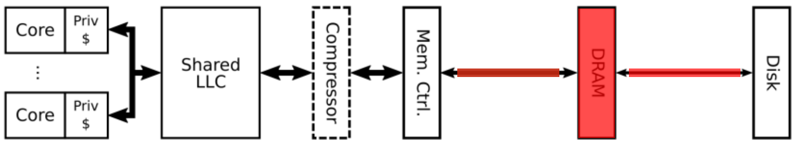
4 2022-02-19

This figure shows the same metrics, for a theoretical system with infinite memory capacity.

As you can see, if we could eliminate *page faults*, performance could be improved by a mean 57% and energy by as much as 48%.



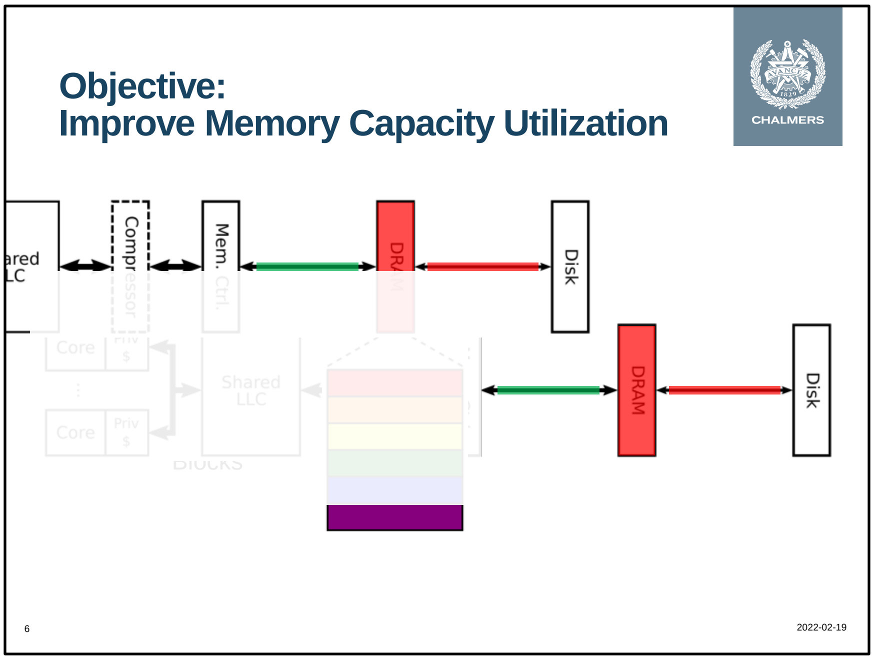
Objective: Improve Memory Bandwidth Utilization



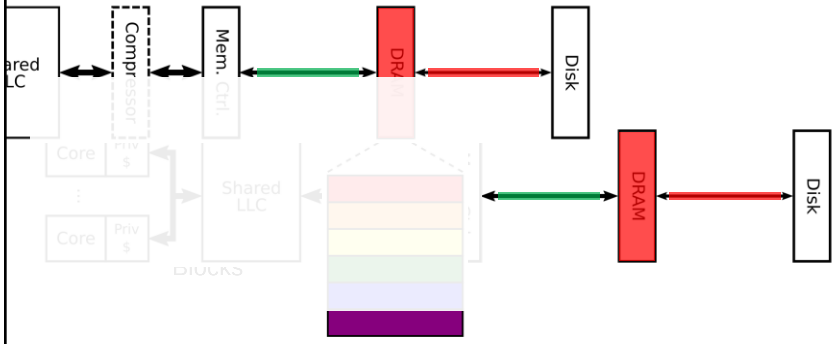
5 2022-02-19

The objective of this thesis is to mitigate these bottlenecks using **compression**. By transferring and storing data in compressed form, the available bandwidth can be utilized more efficiently.

This allows for greater performance and as a side effect reduced energy consumption.

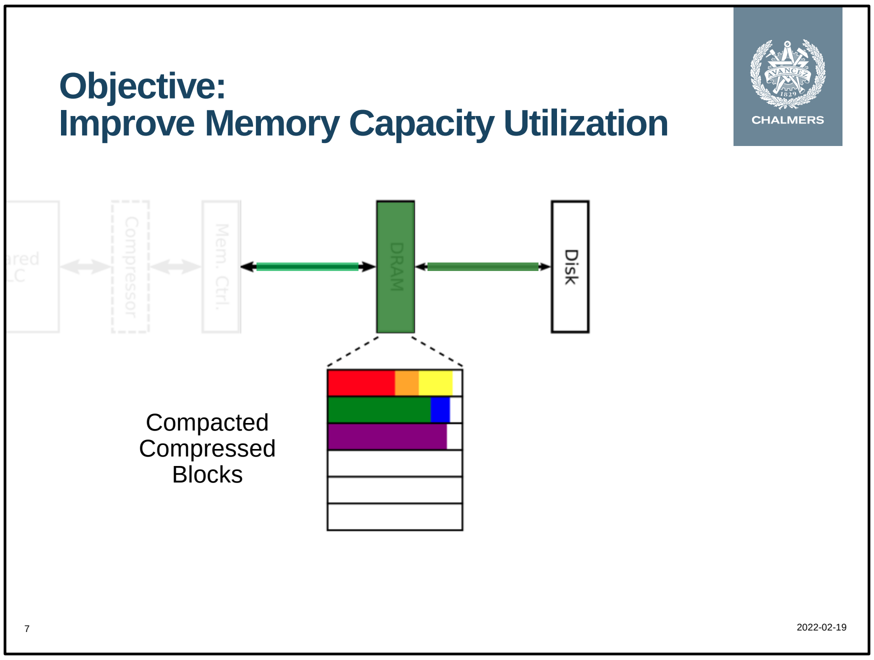


Objective:
Improve Memory Capacity Utilization



6 2022-02-19

Compression alone isn't enough to increase memory capacity, however.



Objective:
Improve Memory Capacity Utilization

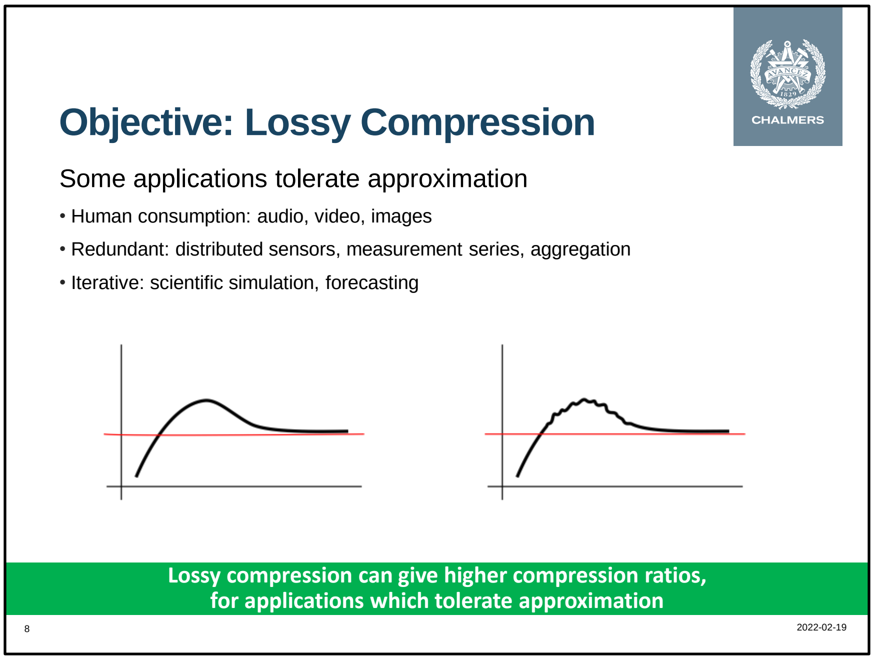
CHALMERS
ared
LC
Compressor
Mem. Ctrl.
DRAM
Disk
Compacted Compressed Blocks

7 2022-02-19

While compressed data will technically use up less space, it isn’t straightforward to make use of this fragmented empty space.

Memory compaction is needed to place the compressed blocks more densely in physical memory, thereby freeing up some of the capacity.

By compacting compressed data in main memory, the effective capacity increases and we can reduce the costly traffic to disk.

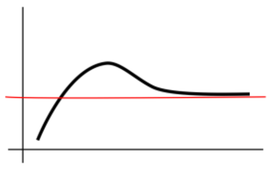


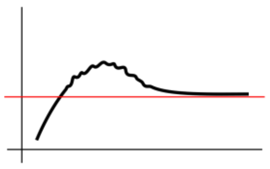
Objective: Lossy Compression

CHALMERS

Some applications tolerate approximation

- Human consumption: audio, video, images
- Redundant: distributed sensors, measurement series, aggregation
- Iterative: scientific simulation, forecasting





Lossy compression can give higher compression ratios, for applications which tolerate approximation

8 2022-02-19

That's all good and well, but memory compression and compaction aren't new ideas.

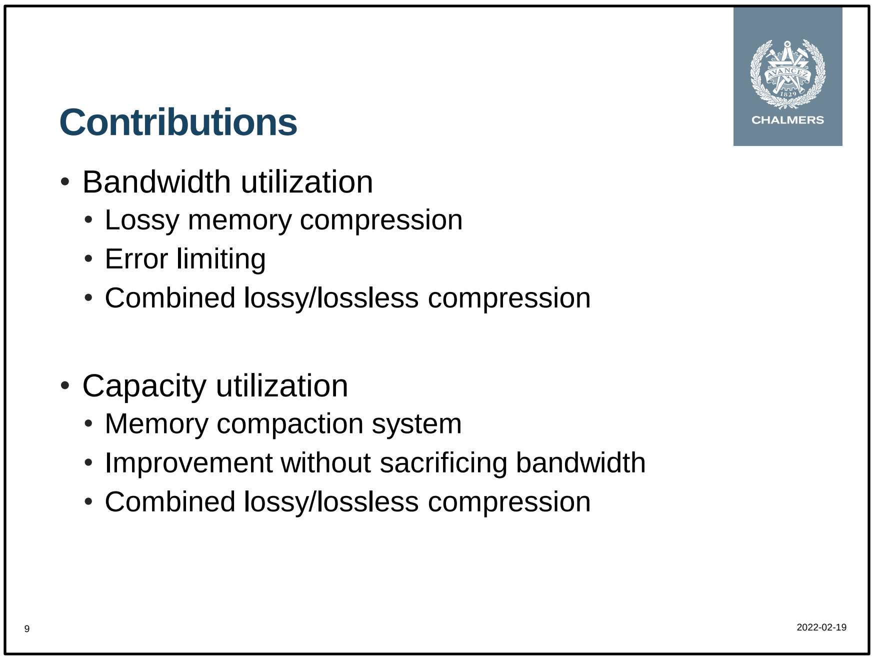
Some classes of applications are able to tolerate **approximation** - precision loss - in their data.

Anything intended for human consumption, such as graphics and audio. We humans forgive all kinds of subtle inaccuracy.

Applications with large amounts of data redundancy, such as aggregation of data from many distributed sensors.

Iterative computation is an especially interesting case. If an application loops over its data and eventually converges to a result, it may be resilient to noise along the way.

Lossy compression is compression which doesn't guarantee perfect reconstruction of the original data. Some amount of approximation is considered acceptable. By exploiting this, higher compression ratios can be achieved.



Contributions

- Bandwidth utilization
  - Lossy memory compression
  - Error limiting
  - Combined lossy/lossless compression

- Capacity utilization
  - Memory compaction system
  - Improvement without sacrificing bandwidth
  - Combined lossy/lossless compression

9 2022-02-19

The contributions of this thesis can be divided up as follows:

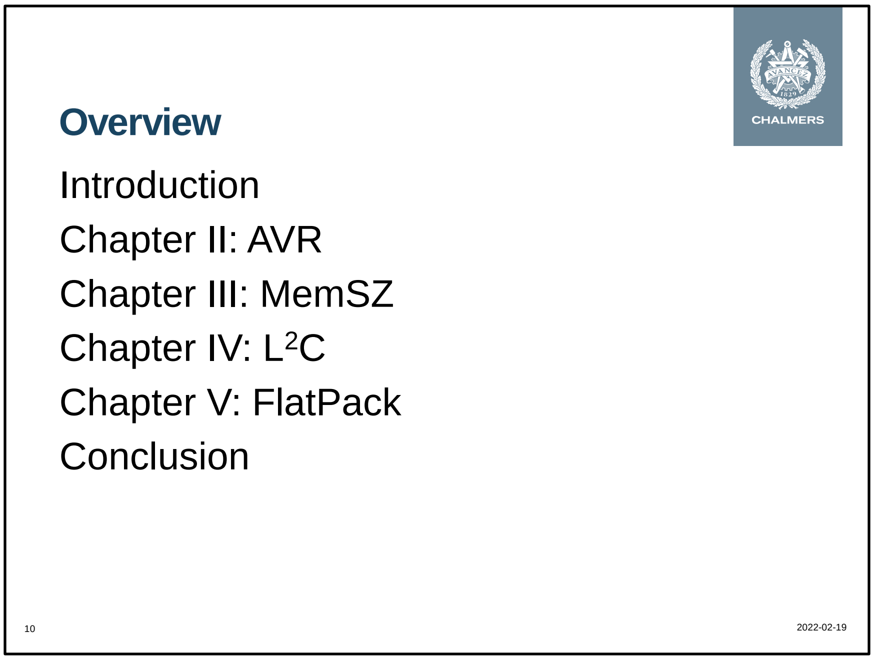
To improve bandwidth utilization, Lossy compression is applied to memory traffic.

To control the approximation effects of this compression, an error limiting mechanism is introduced
Finally, since lossy compression is only applicable to a subset of data in a subset of applications, techniques are presented to combine lossy and lossless compression.

To improve memory capacity, I build on that work:
Chapter 5 introduces a novel memory compaction mechanism which works in harmony with the lossy memory compression.

The main feature of this system is that, unlike previous designs, it is able to improve memory capacity without sacrificing bandwidth.

Finally, this memory compactor is extended to support the combined lossy/lossless compression.



Overview

Introduction

Chapter II: AVR

Chapter III: MemSZ

Chapter IV: L^2C

Chapter V: FlatPack

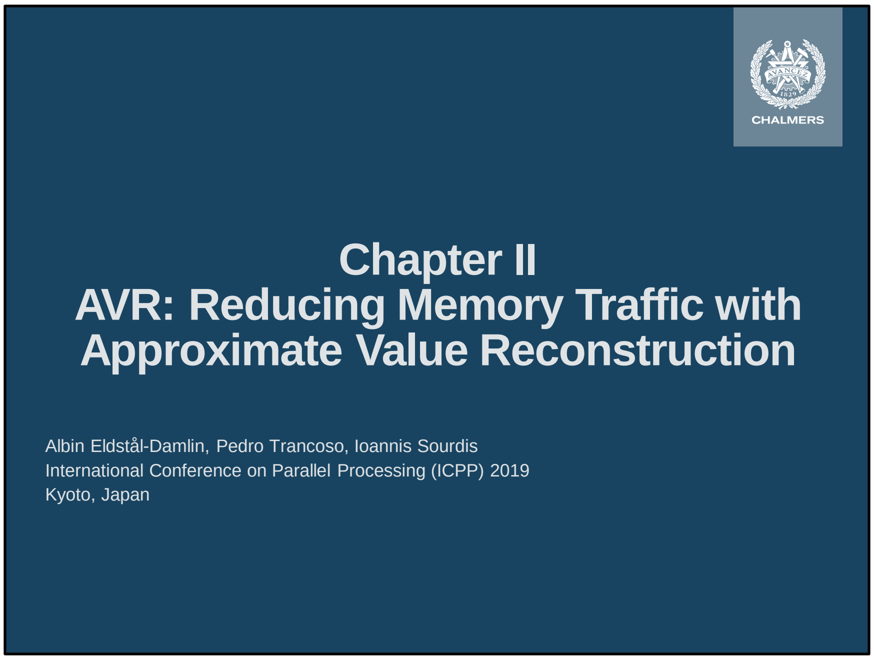
Conclusion

10

2022-02-19

The rest of this presentation will follow this format.

After this brief introduction, I will cover each of the main chapters of the thesis before the concluding remarks.

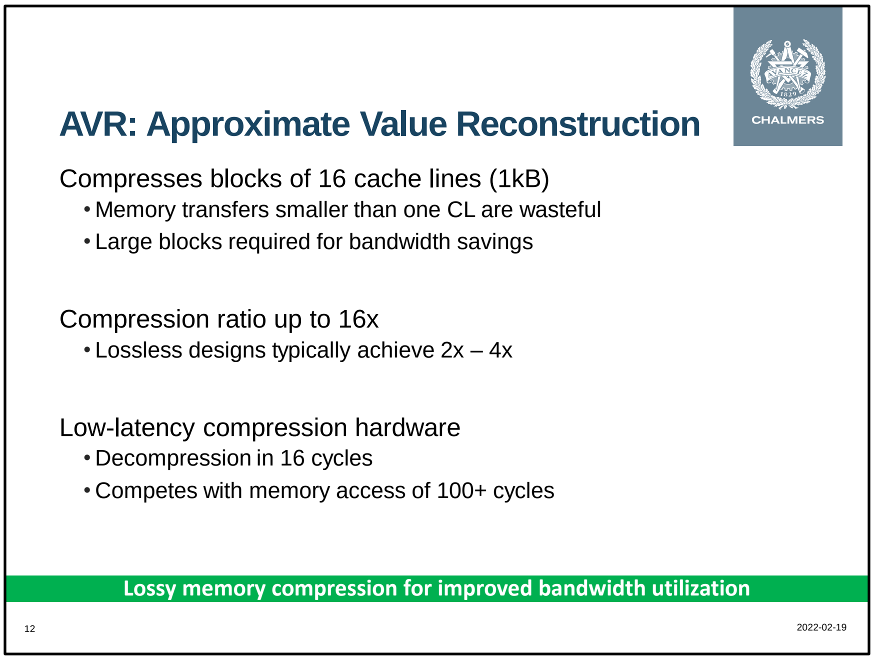


CHALMERS

Chapter II
AVR: Reducing Memory Traffic with Approximate Value Reconstruction

Albin Eldstål-Damlin, Pedro Trancoso, Ioannis Sourdis
International Conference on Parallel Processing (ICPP) 2019
Kyoto, Japan

The second chapter of the thesis describes AVR: Approximate Value Reconstruction. This was originally presented at ICPP 2019.



AVR: Approximate Value Reconstruction

Compresses blocks of 16 cache lines (1kB)
- Memory transfers smaller than one CL are wasteful
- Large blocks required for bandwidth savings

Compression ratio up to 16x
- Lossless designs typically achieve 2x – 4x

Low-latency compression hardware
- Decompression in 16 cycles
- Competes with memory access of 100+ cycles

Lossy memory compression for improved bandwidth utilization

12 2022-02-19

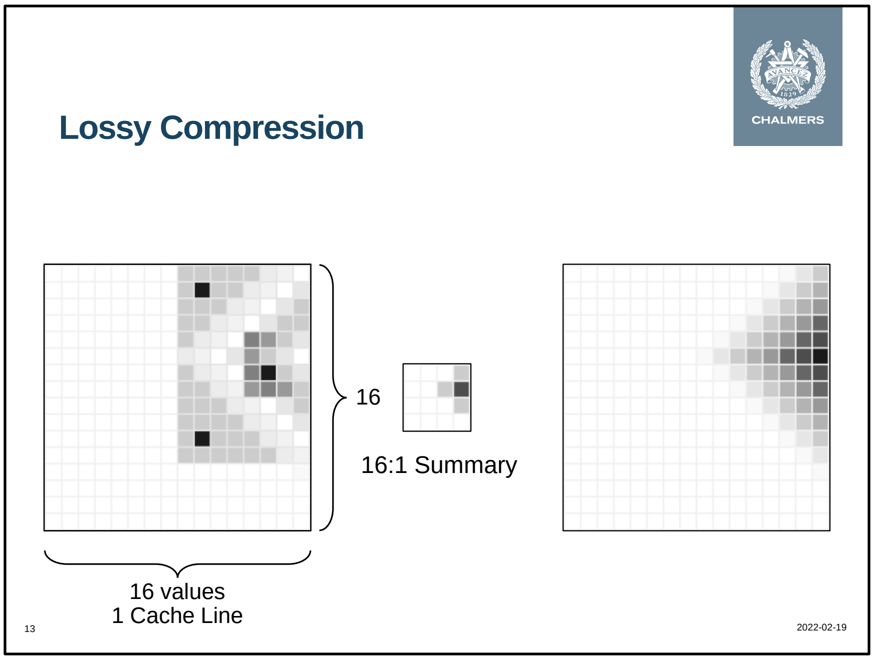
AVR is a lossy memory compression system, which targets the bandwidth bottleneck.

Main memory is typically accessed at the granularity of a single cache line. Therefore, compressing single cache lines will not save bandwidth. For this reason, AVR compresses large blocks of 16 cache lines together.

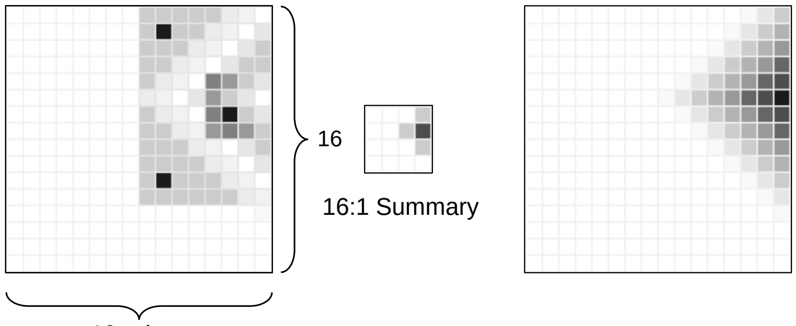
This, combined with the more aggressive lossy compression, allows AVR to reach compression ratios of up to 16x. By comparison, lossless methods typically land between 2x and 4x.

Decompression latency is critical to performance, since decompression is on the critical path of memory reads. For this reason, AVR uses a simple compression scheme, where decompression takes 16 cycles. This is significantly shorter than the latency of a memory access.

In short, AVR uses lossy memory compression to improve bandwidth utilization.



Lossy Compression



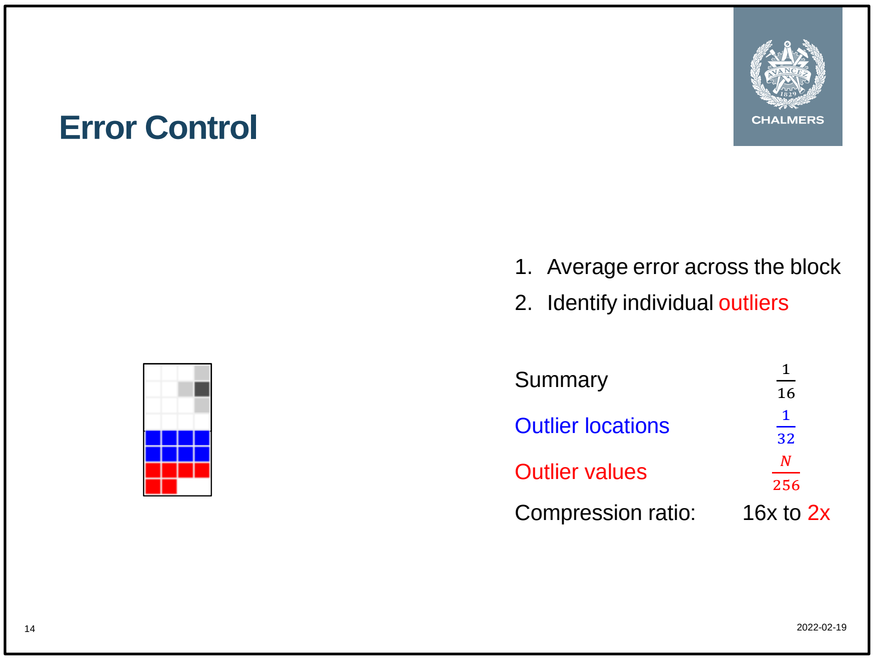
13 2022-02-19

The compression method used by AVR is known as **downsampling**. On the screen, you see our 16 cache lines stacked on top of one another, as 256 numeric values. The shading of each small square represents a single value.

To compress this block, we can compute the average of each segment of the block, like so. As you can see, lighter regions give a light average and darker regions give a dark average. We call this the **summary** of a block, and it is exactly one sixteenth the size of the original.

To decompress, we simply interpolate between the averages in the summary, back to the original size of 16x16. This can be done in 16 CPU cycles, since the operation is quite open to parallelization.

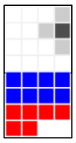
As you can see if we look at the difference between the two, the reconstruction isn't perfect. This brings us to the main challenge of lossy compression: controlling the approximation error.



Error Control

CHALMERS

1. Average error across the block
2. Identify individual outliers



Summary	$\frac{1}{16}$
Outlier locations	$\frac{1}{32}$
Outlier values	$\frac{N}{256}$
Compression ratio:	16x to 2x

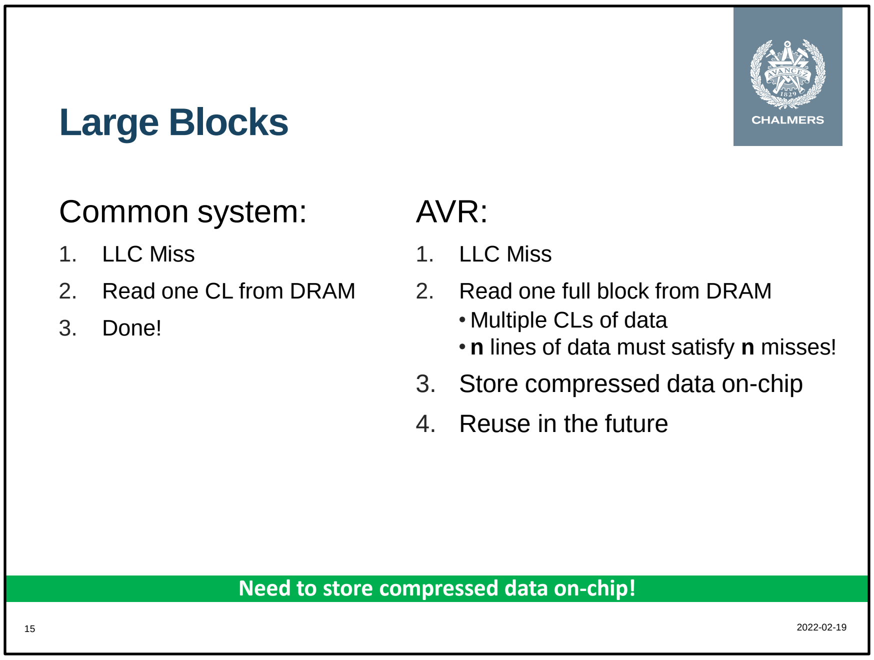
14 2022-02-19

AVR deals with the errors of interpolation in two distinct ways. First, whenever a block is compressed, we calculate the average error across the block. If it is found to exceed an acceptable level, the block is left uncompressed.

Second, we can identify individual values which suffer excessive error. In the example, these three here. We call these values OUTLIERS. If we store these outlier values explicitly, their error can be avoided the next time the block is needed.

So, to recap. We've computed a 16:1 summary of our block, which represents most of the values well enough. If they are all within acceptable limits, we have a compression ratio of 16x.

If, on the other hand, we identified any outliers, we store them alongside the summary, with enough information to put them back. This way, the compression ratio falls gradually down to 2x.



Large Blocks

Common system:

1. LLC Miss
2. Read one CL from DRAM
3. Done!

AVR:

1. LLC Miss
2. Read one full block from DRAM
   - Multiple CLs of data
   - **n** lines of data must satisfy **n** misses!
3. Store compressed data on-chip
4. Reuse in the future

Need to store compressed data on-chip!

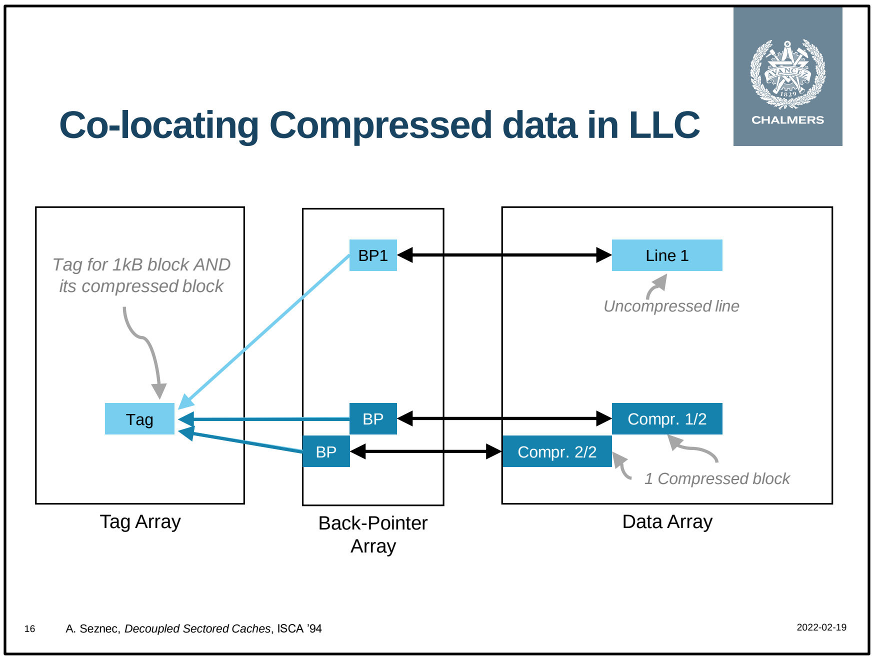
15 2022-02-19

Compressing large blocks introduces a new challenge. When multiple cache lines are compressed together this way, they become dependent on each other, and random access within a block is made impractical.

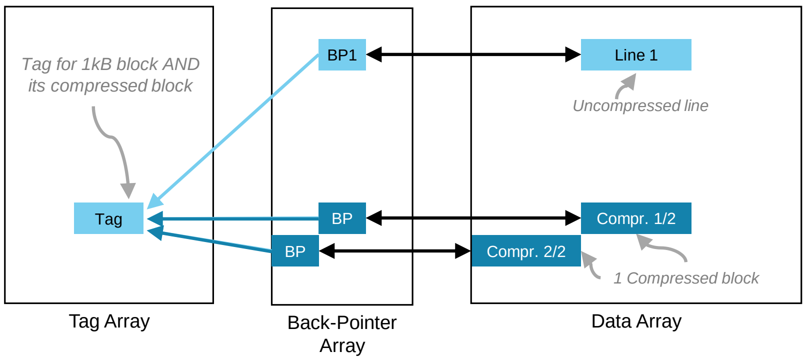
In a normal cache hierarchy, miss in the Last Level Cache is easy to resolve. The missing cache line is read from main memory and that's it.

In a system like AVR, the entire compressed block must be fetched from main memory. This compressed block may be larger than a single cache line, which leads to a traffic overhead. If the compressed block has a size of N cache lines, this single fetch must satisfy N future LLC misses, otherwise we've actually INCREASED the total traffic to main memory! AVR solves this challenge by keeping the compressed block on-chip after the miss has been resolved. It is likely, due to locality, that the other cache lines compressed into the same block will be reused in the future.

To support this, AVR needs an LLC which is able to store compressed blocks of multiple lines.



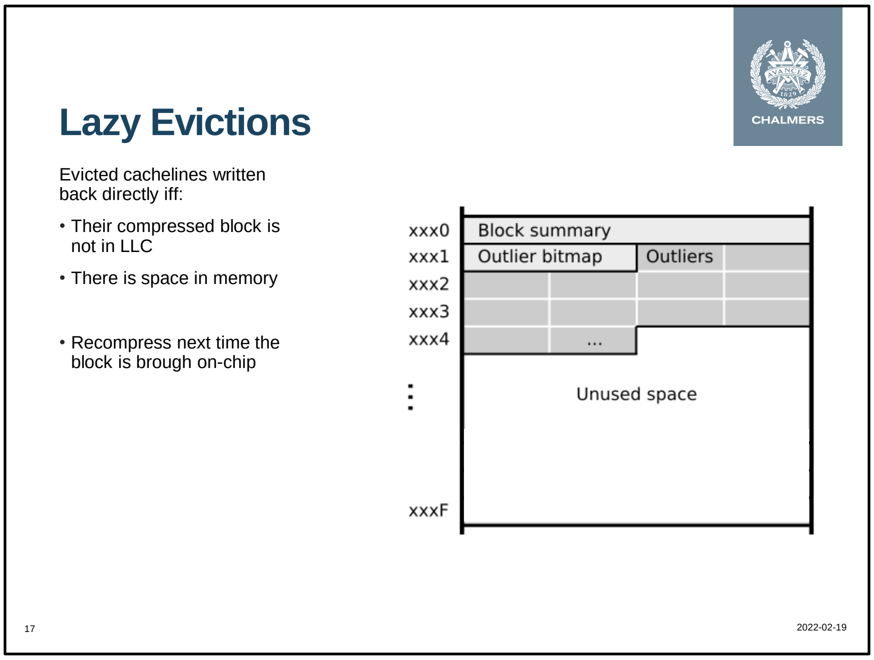
Co-locating Compressed data in LLC



The way we achieve this is using an existing technique known as a Decoupled Sectored Cache. It allows multiple cache lines to be associated with the same tag. This is done by introducing a layer of indirection known as the Back Pointer Array. Each data entry has its own corresponding Back Pointer in this array. The purpose of the back-pointers is to establish a many-to-one relationship between data entries and tags.

AVR modifies this design so that a single tag is responsible for a 1kB block of memory, in both compressed and uncompressed form. Any combination of compressed and uncompressed data representing the same block can be present in the LLC at the same time.

In experiments, between 2 and 16% of the AVR LLC is used for compressed data.

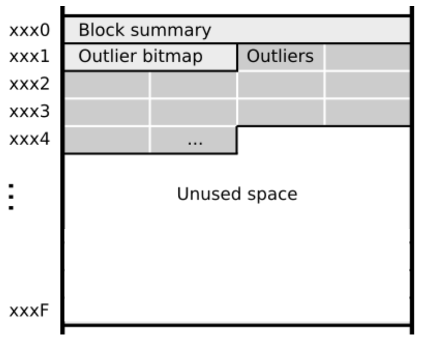


Lazy Evictions

Evicted cachelines written back directly iff:

- Their compressed block is not in LLC
- There is space in memory

- Recompress next time the block is brough on-chip



17

2022-02-19

Another problem with large blocks is that we need all 16 cache lines in order to recompress.

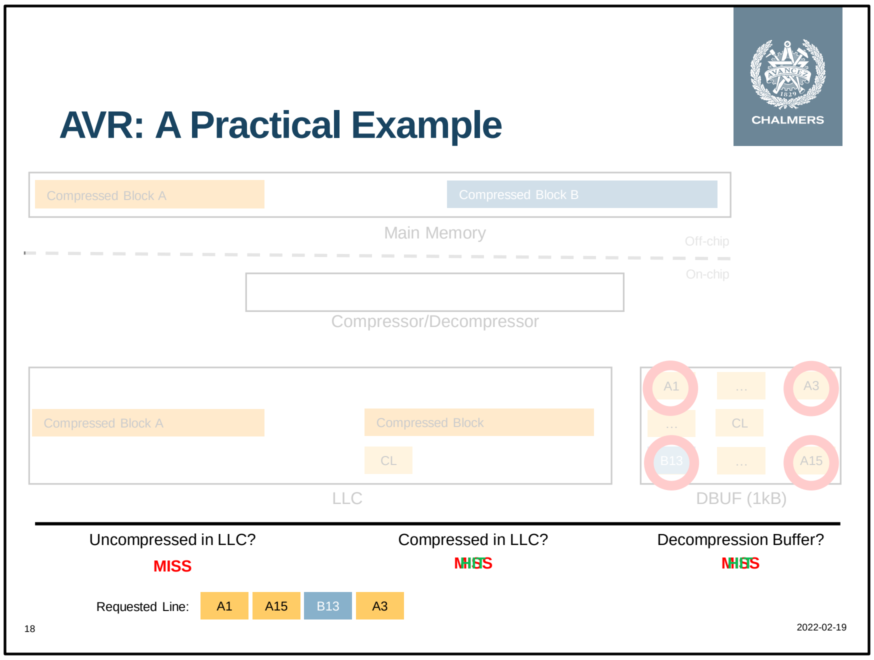
What happens if a single dirty cache line is on chip, and needs to be written back, but the rest of the block is only available in main memory?

AVR stores its compressed blocks without compacting them in memory, which means there is empty space trailing each block.

If there is enough space there for one cache line, the dirty line is written back as-is. This is called a **lazy eviction**.

The next time the block is needed on-chip, it is brought in together with the lazily evicted cache line.

This delays the costly memory read until the next miss, thus reducing the traffic overhead.



Putting all these parts together, we have an architecture for lossy memory compression. Let's look at a functional example.

We're going to emulate an AVR system, with the main memory up here at the top. It contains two compressed blocks, A and B. Below that, on-chip, we have our hardware compressor, and the decoupled LLC discussed earlier. Alongside the LLC, we have a Decompression buffer, DBUF, which is used to temporarily store the most recently decompressed block. It is 1kB, to precisely fit the one block.

When a request to the LLC comes in, we perform three lookups in parallel:

First, the requested line may be in the LLC in its uncompressed form. Return it.

Second, the block the line belongs to may be present in its compressed form. Decompress the block to get the data.

Third, the line may belong to the most recently decompressed block, which means it is present in the decompression buffer.

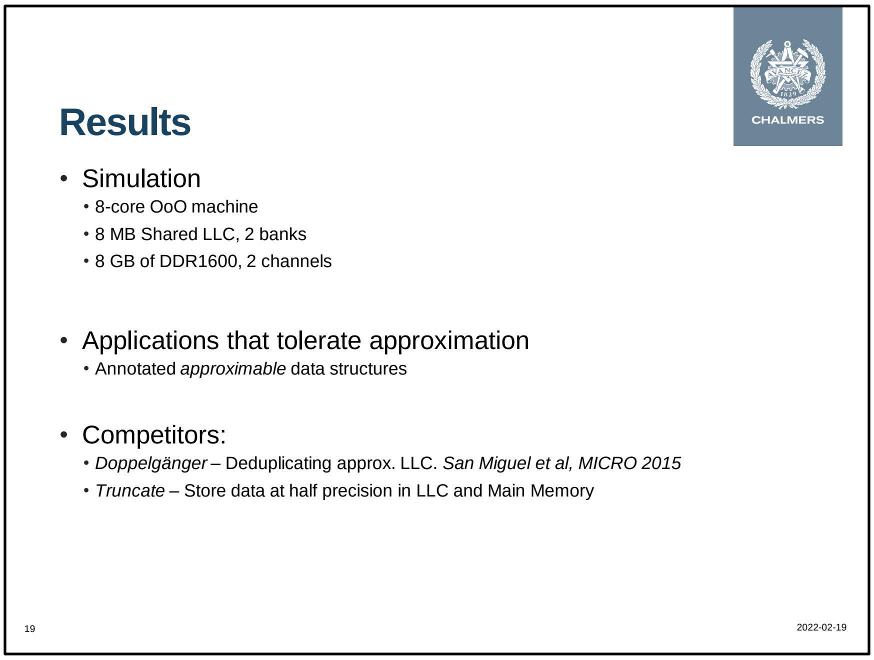
Here's an example of a request, A1 which is the first of the 16 cachelines of block A. Our three-way lookup comes back MISS-MISS-MISS, so we have to go to memory. We bring in the compressed block and decompress it into DBUF. At this point, we have the data that was requested and can return it immediately. Request served, Processor core keeps chooching. In the background, we anticipate reuse both of this line and this block; both are inserted in LLC.

Another request, A15. Our three-way lookup yields MISS-HIT-HIT. We have the compressed block A, but more importantly we have the entire block uncompressed in DBUF. A15 is right there and we can return it. Just like before, we insert A15 in LLC in anticipation of reuse.

The next request is for B13, which belongs to block B. Three-way lookup comes back MISS-MISS-MISS, so we must go to memory. As before, we bring in the block and feed it to our decompressor. Once decompression completes, the data must go in DBUF, so we purge the old data from there and overwrite it. The requested line is available now. Serve the request, put it and the compressed block in LLC for future reuse. Move on.

Finally, we see another request for a line in A. MISS-HIT-MISS, we have only the compressed block on chip. No problem, read it out of LLC and feed it into the decompressor. Decompress into DBUF. Serve the request. Insert in LLC.

After these four requests, we've only gone to memory twice. In return, we now have 32 cachelines of data in LLC ready for use!



Results

CHALMERS

- Simulation
  - 8-core OoO machine
  - 8 MB Shared LLC, 2 banks
  - 8 GB of DDR1600, 2 channels
- Applications that tolerate approximation
  - Annotated *approximable* data structures
- Competitors:
  - *Doppelgänger* – Deduplicating approx. LLC. *San Miguel et al, MICRO 2015*
  - *Truncate* – Store data at half precision in LLC and Main Memory

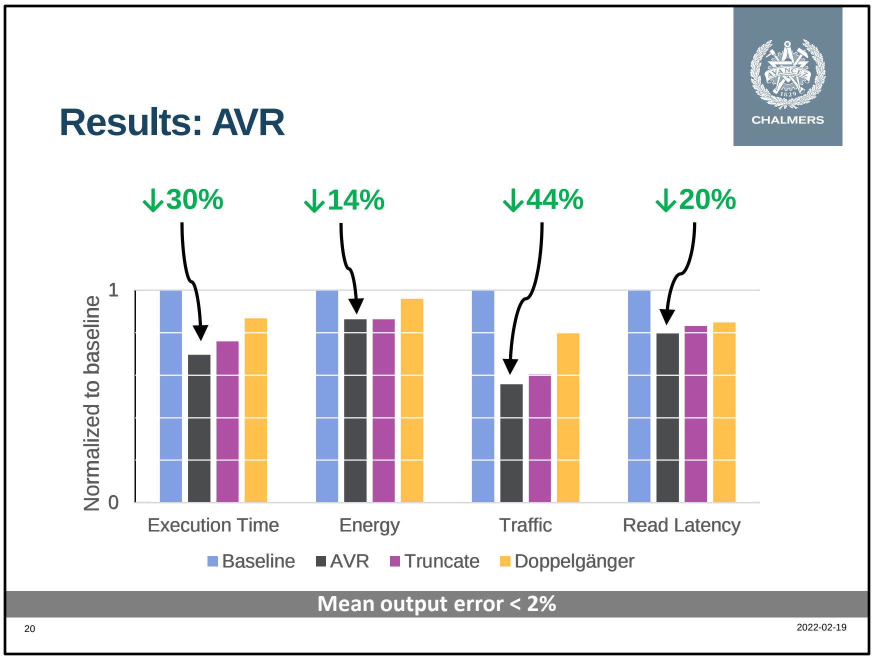
19 2022-02-19

To evaluate AVR, I've implemented it in a processor and memory simulator, simulating an 8-core machine with a banked 8MB LLC.

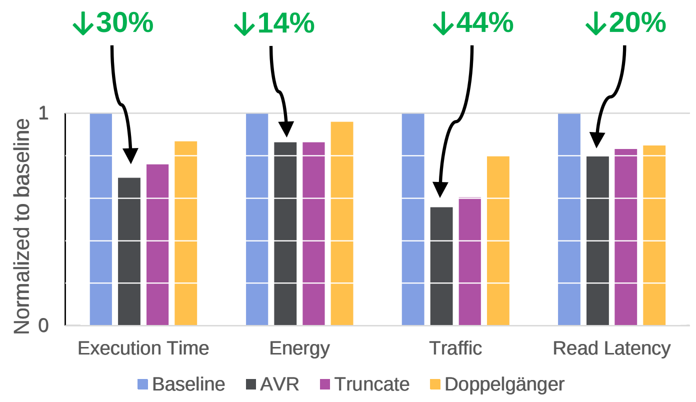
I identified benchmarks which could tolerate approximation and annotated their source code to mark parts of their dataset as approximable.

I will show results compared to two competitive designs:

- **Doppelgänger** (MICRO 2015) is an approximate deduplicating LLC which identifies approximately similar cachelines and stores only one of them.
- Truncate applies a simple precision reduction, cutting the footprint of approximable data in half both on the memory bus and in the LLC.



Results: AVR



Mean output error < 2%

20 2022-02-19

This figure shows the simulation results for four important metrics. Lower is better.

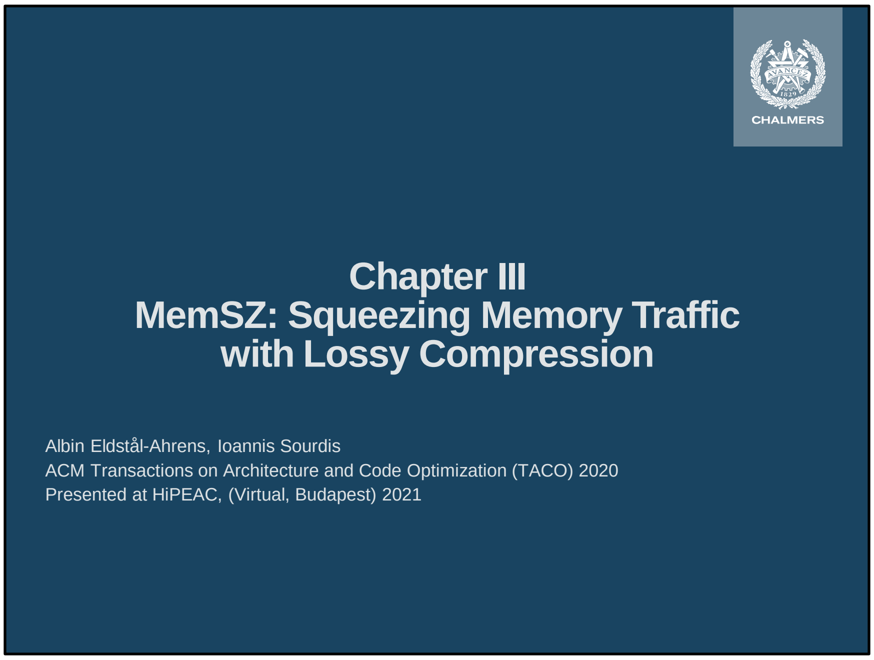
AVR improves performance, reducing the mean execution time by 30%.

This reduced execution time also gives energy benefits, with system energy consumption reduced by 14%.

These benefits stem from a reduction in memory traffic, by an average of 44%.

The average latency of a read operation is reduced by an average of 20%. This is due in part to an improved LLC hit rate.

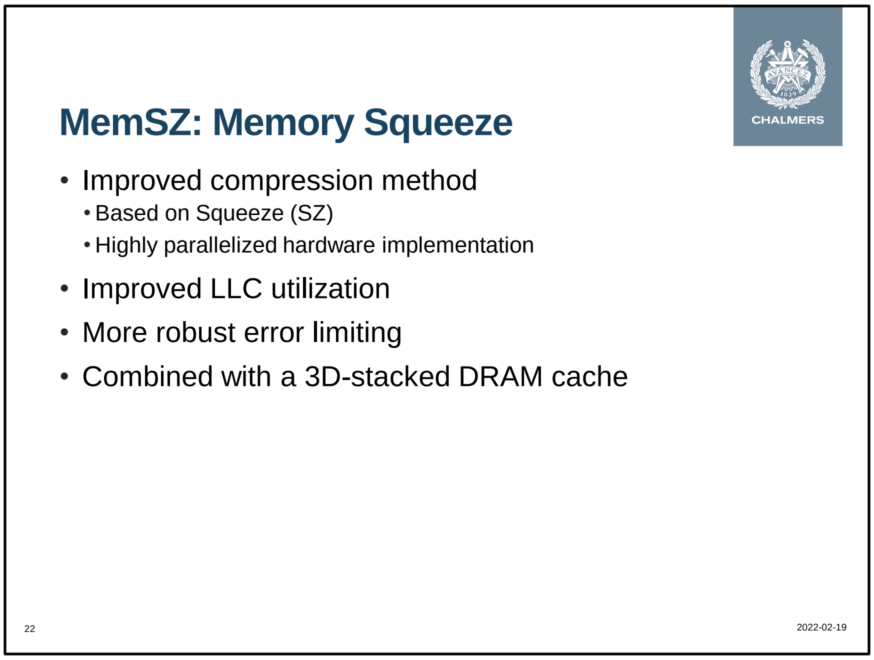
Finally, my simulations also include the quality degradation introduced by the lossy compression. When comparing the application output to the uncompressed baseline, it deviates on average by less than 2%.



Chapter III
MemSZ: Squeezing Memory Traffic with Lossy Compression

Albin Eldstål-Ahrens, Ioannis Sourdis
ACM Transactions on Architecture and Code Optimization (TACO) 2020
Presented at HiPEAC, (Virtual, Budapest) 2021

The next chapter outlines MemSZ. This was published as an original paper in TACO 2020 and presented at HiPEAC 2021.


MemSZ: Memory Squeeze

- Improved compression method
  - Based on Squeeze (SZ)
  - Highly parallelized hardware implementation
- Improved LLC utilization
- More robust error limiting
- Combined with a 3D-stacked DRAM cache

22 2022-02-19

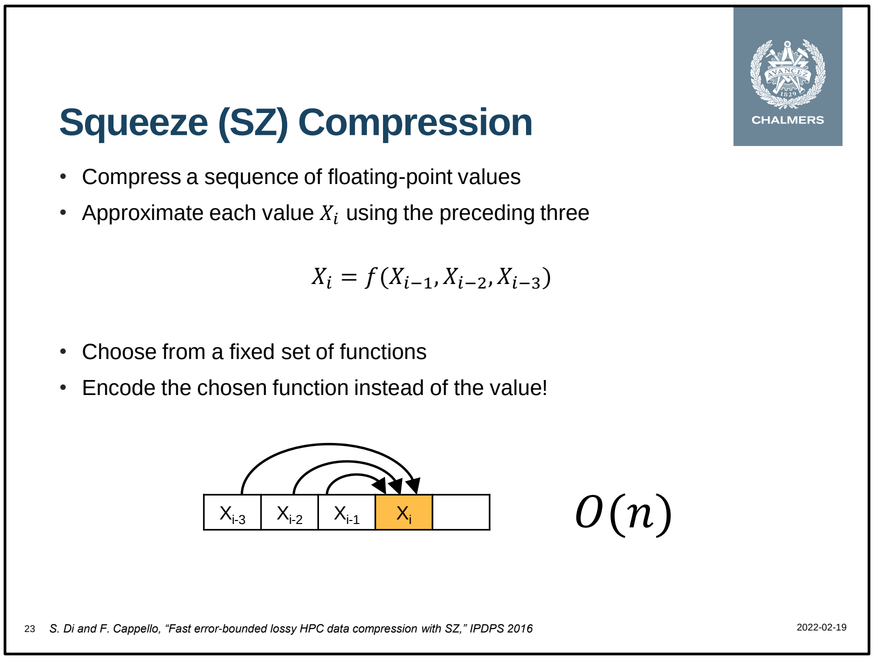
Memory Squeeze extends on the design of AVR in four key ways.

First, an improved compression method, based on an existing algorithm called SZ. MemSZ provides a novel parallelized hardware implementation, suitable for memory compression.

Second, an optimization to improve the space efficiency of the LLC

Third, a more robust, global mechanism to limit approximation error

Finally, MemSZ can be more easily integrated into systems equipped with a 3D-stacked DRAM cache.

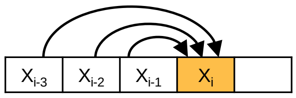


Squeeze (SZ) Compression

- Compress a sequence of floating-point values
- Approximate each value X_i using the preceding three

$$X_i = f(X_{i-1}, X_{i-2}, X_{i-3})$$

- Choose from a fixed set of functions
- Encode the chosen function instead of the value!



$O(n)$

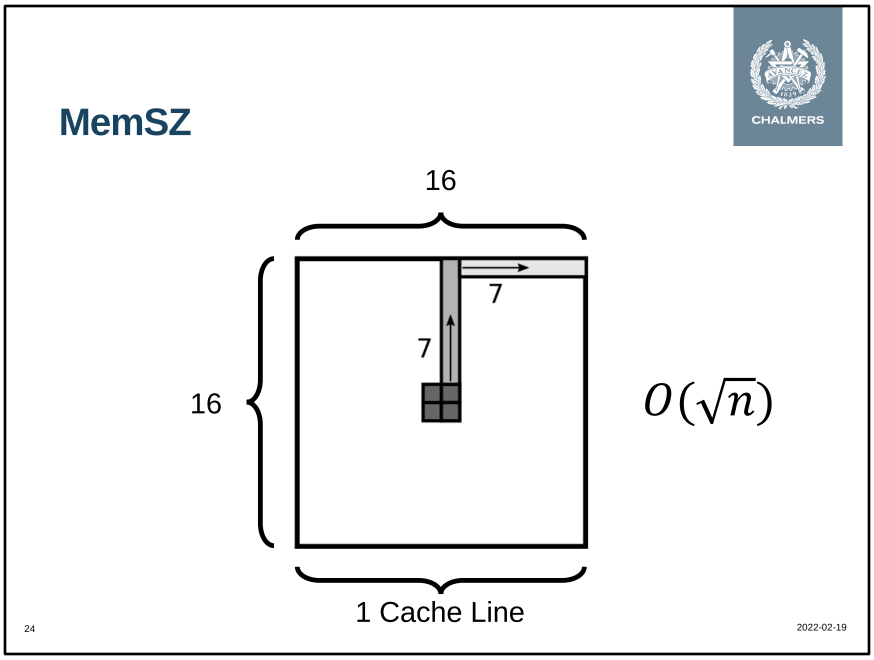
SQUEEZE, introduced at IPDPS in 2016, is a lossy compression method, which works on sequences of numbers.

The idea is to describe each number not as a binary value, but as a function of the preceding values.

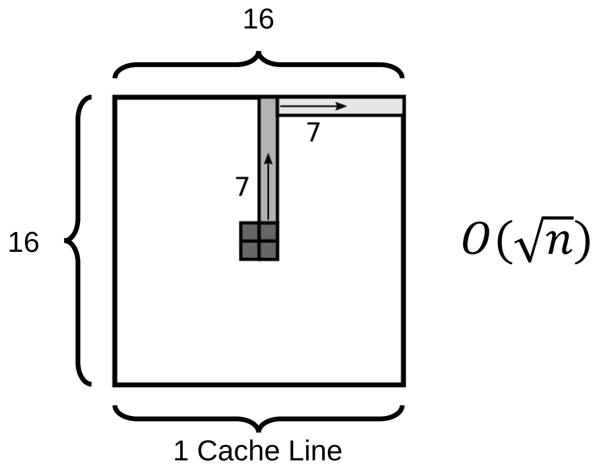
With a limited number of functions, there are a limited number of options and therefore less information to store.

This algorithm has a trivial dependency chain, such that each individual value can be decompressed only when the preceding values have been.

As a result, squeeze decompression has linear complexity.



MemSZ



24

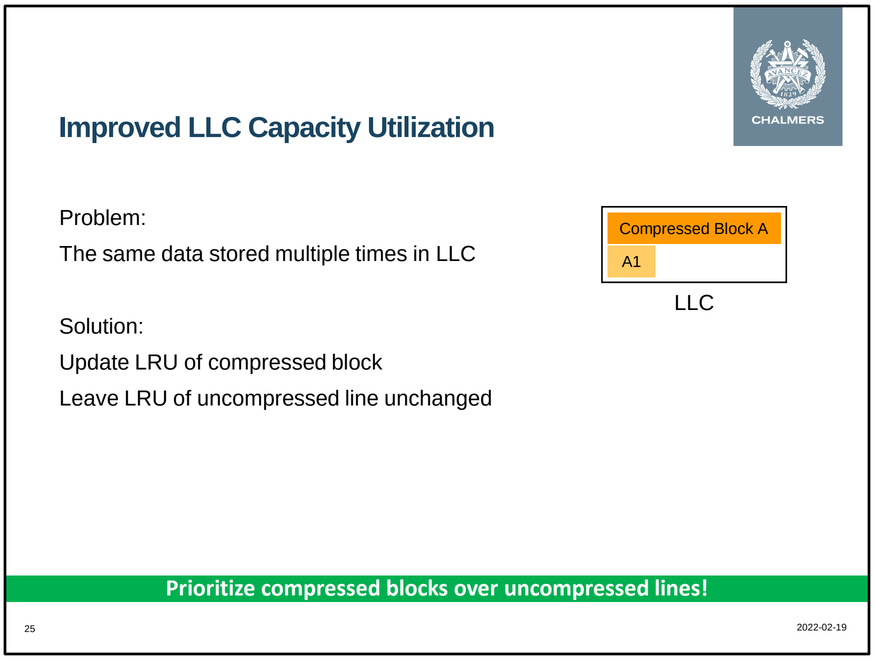
2022-02-19

MemSZ introduces a highly parallelized hardware adaptation of Squeeze, which works as follows:

First, we take our 16 cache lines (256 values) and arrange them in a square, like so. Then, we select a few **starting values** for our sequences. We choose the four values right in the center of the block.

From each pair of starting values, we can start processing both up and down in parallel, giving us these four **struts** running vertically down the middle of the block. From there, we can parallelize even more.

Once the struts are ready, we can start 32 parallel **arms** spreading out toward the edges of the block. Normal squeeze would have had to process all 256 values in one long sequence. With this technique, the critical path is one strut of seven and one arm of seven. The complexity is reduced by the square root!



Improved LLC Capacity Utilization

CHALMERS

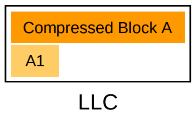
Problem:

The same data stored multiple times in LLC

Solution:

Update LRU of compressed block

Leave LRU of uncompressed line unchanged



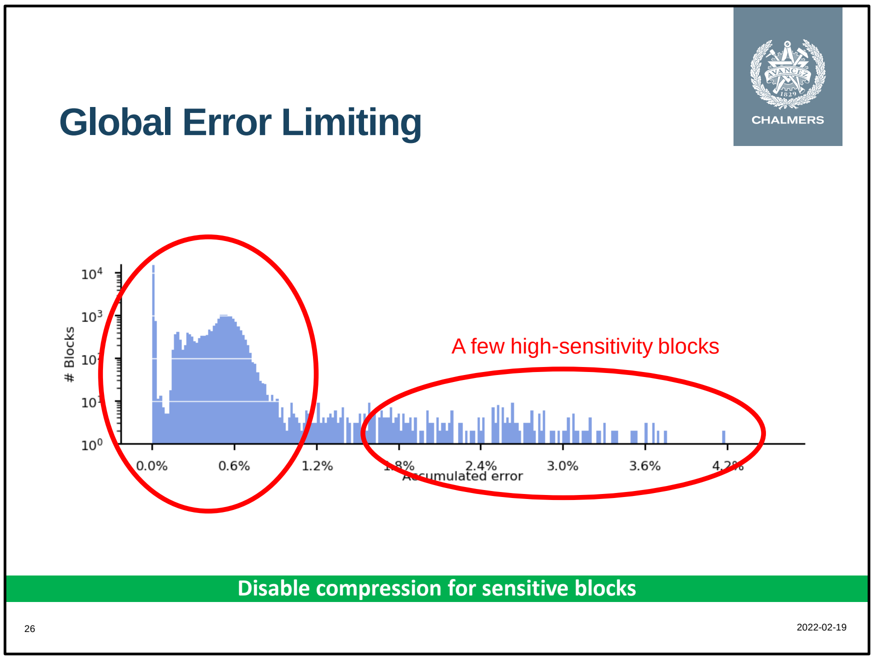
Prioritize compressed blocks over uncompressed lines!

25 2022-02-19

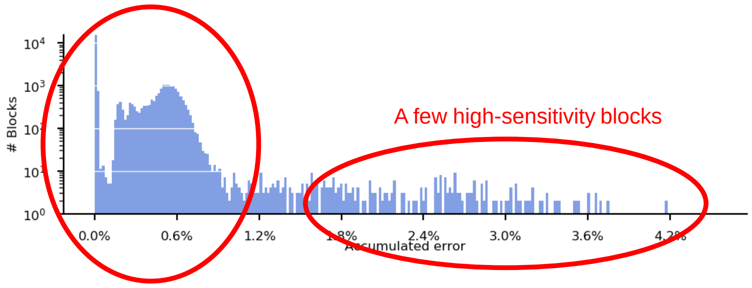
The second contribution is an optimization to the space efficiency of the decoupled last-level cache. As you may recall from earlier, we can end up storing both an uncompressed cache line, like A1 here, and the compressed block it belongs to at the same time. This redundancy amounts to pollution of the LLC.

AVR uses an Least-Recently-Used replacement policy, and updates a counter whenever a cache line is accessed. When a line needs to be replaced, the least recently used one is thrown out.

MemSZ solves the pollution problem using these counters. When the uncompressed line is accessed, the LRU counter for the compressed block is updated in stead. This way, when it comes time to replace data in the cache, it is more likely that the uncompressed line be replaced than the compressed block.



Global Error Limiting



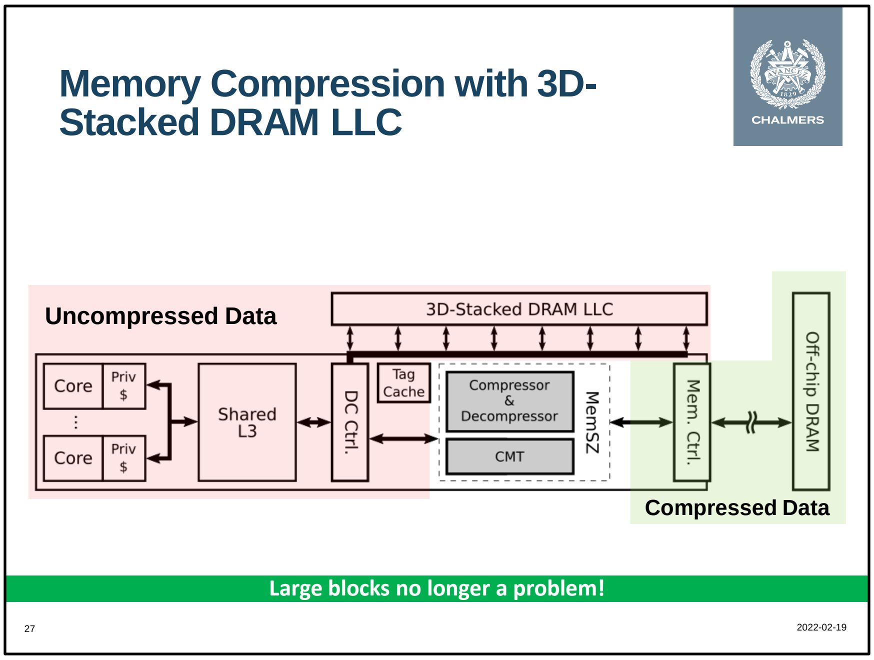
Disable compression for sensitive blocks

26 2022-02-19

The third contribution relates to error limiting. This is a histogram over the total accumulated errors per block over the entire execution of an application. As you can see, the vast majority of blocks are collected near zero, meaning they accumulated only a small error over the entire execution.

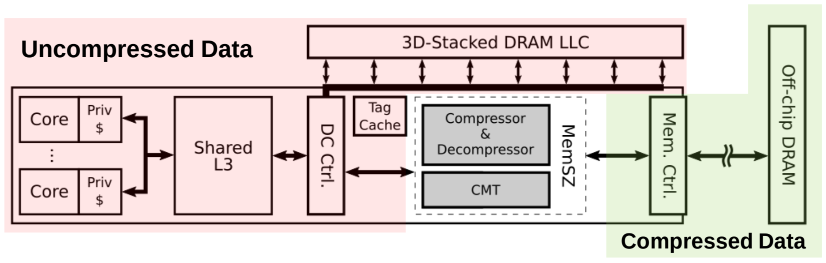
There is, however, a long tail of blocks with higher accumulated errors. These blocks may cause disproportionate effects in the quality of the output. It is therefore a valuable performance/quality trade-off to disable compression for these specific blocks, if we can identify them at runtime.

MemSZ maintains an accumulating error counter for each individual block and has a threshold above which compression is permanently disabled. This has the effect of cutting that long tail of sensitive blocks.



Memory Compression with 3D-Stacked DRAM LLC

CHALMERS



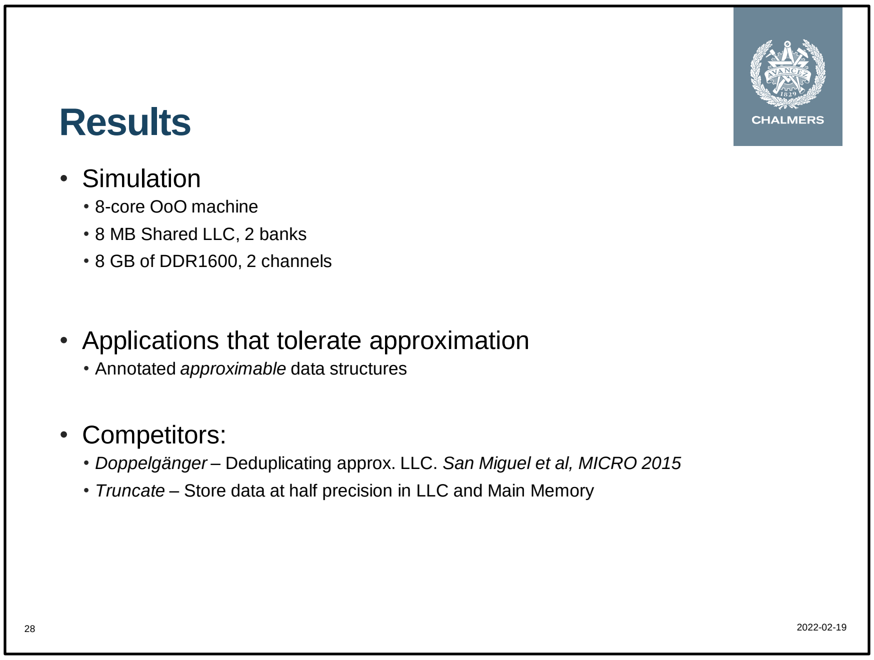
Large blocks no longer a problem!

27 2022-02-19

The fourth and final contribution is an alternative architecture for systems equipped with 3D-stacked DRAM. This is a module of high-bandwidth DRAM attached directly to the processor package, with a size somewhere between the normal SRAM caches and main memory.

MemSZ with a DRAM LLC is a natural extension to the design we've seen so far, with one crucial difference. No compressed data is stored on-chip. By employing a 1kB line size in the DRAM cache, full uncompressed blocks fit perfectly.

One effect of this is that the SRAM L3 cache doesn't need to store compressed data anymore. Another consequence is that our large block size no longer causes any memory traffic overhead, since an uncompressd LLC miss now requires 1kB of data.



Results

- Simulation
  - 8-core OoO machine
  - 8 MB Shared LLC, 2 banks
  - 8 GB of DDR1600, 2 channels
- Applications that tolerate approximation
  - Annotated *approximable* data structures
- Competitors:
  - *Doppelgänger* – Deduplicating approx. LLC. *San Miguel et al, MICRO 2015*
  - *Truncate* – Store data at half precision in LLC and Main Memory

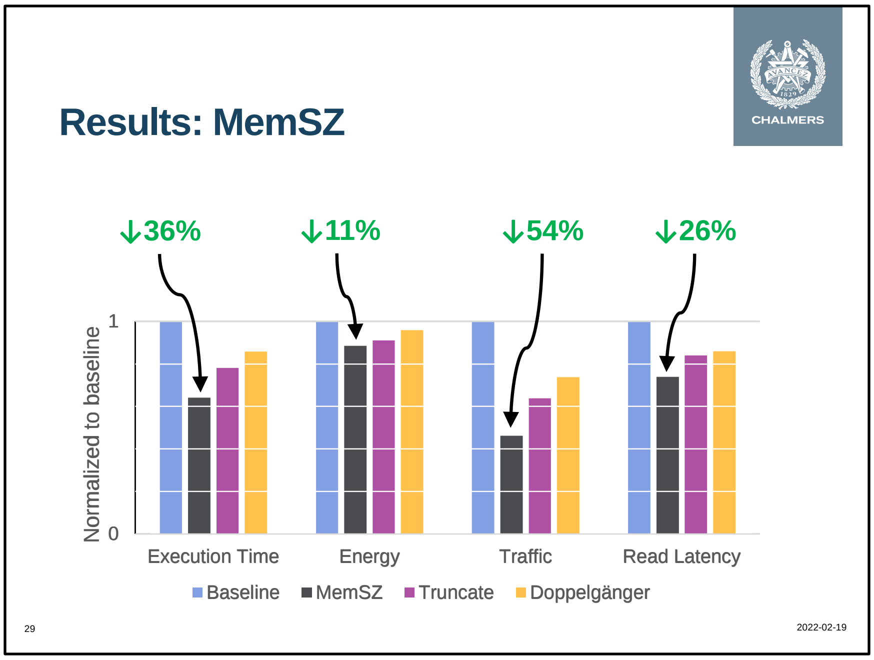
28

2022-02-19

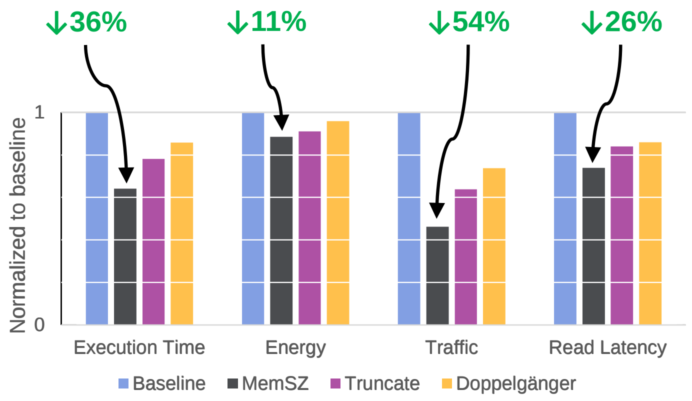
Finally, let's talk about the impact of MemSZ on system performance.

As with AVR, I’ve implemented MemSZ in a processor simulator.

We will see how it compares to the same two competing designs: Doppelgänger and Truncate.



Results: MemSZ



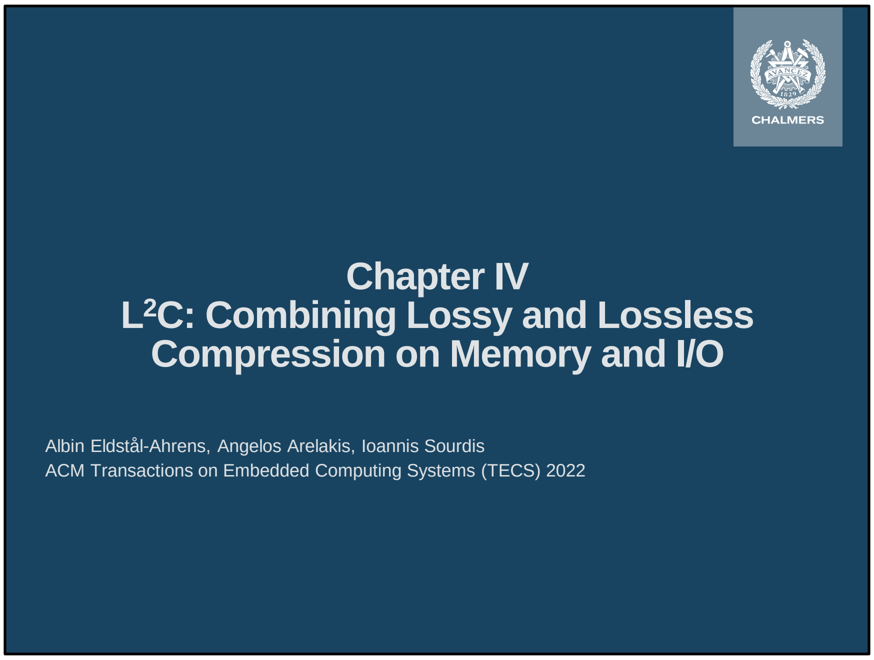
As before, lower is better for these metrics.

MemSZ reduces mean benchmark execution time by 36%.

This leads to an energy consumption 11% lower than the baseline.

Memory traffic saw a mean reduction of 54%.

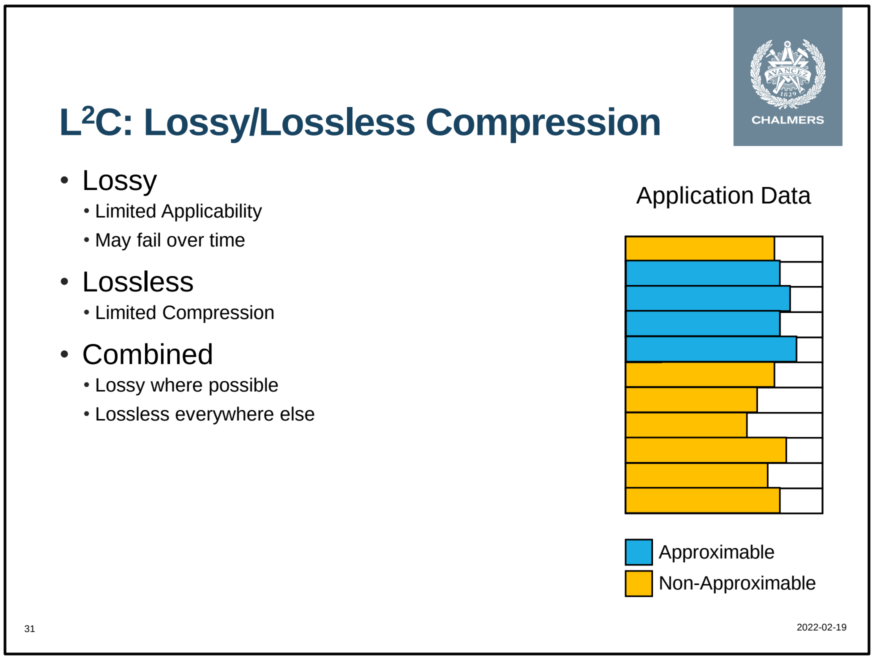
The average latency of a read operation was reduced by 26%.



Chapter IV
L^{2}C: Combining Lossy and Lossless Compression on Memory and I/O

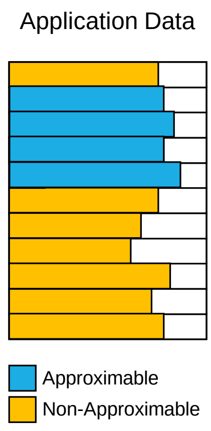
Albin Eldstål-Ahrens, Angelos Arelakis, Ioannis Sourdis
ACM Transactions on Embedded Computing Systems (TECS) 2022

The fourth chapter presents L2C, which was published as a journal paper in TECS this year.



L^{2}C: Lossy/Lossless Compression

- Lossy
  - Limited Applicability
  - May fail over time
- Lossless
  - Limited Compression
- Combined
  - Lossy where possible
  - Lossless everywhere else



31

2022-02-19

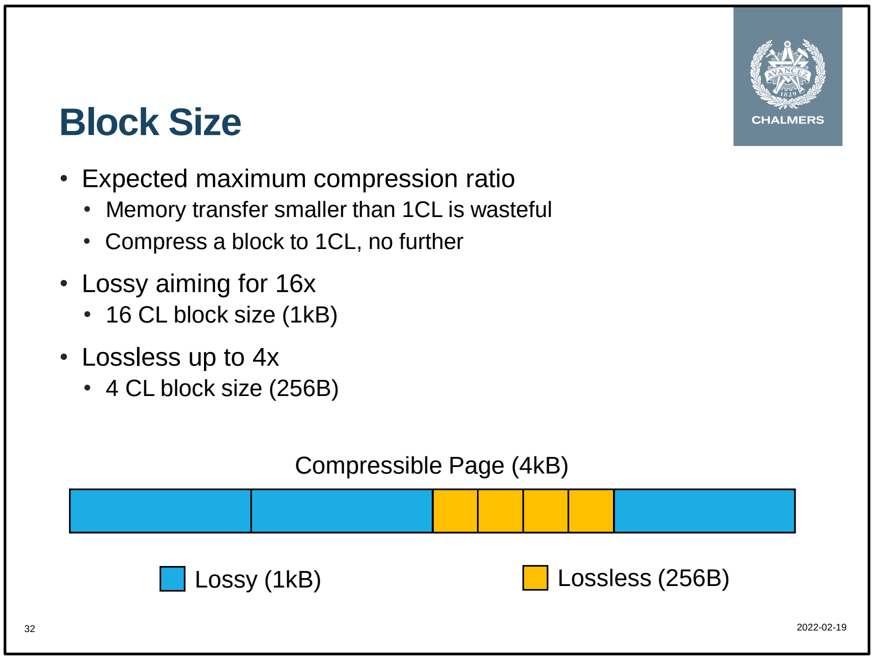
Lossy memory compression has two main shortcomings. Let’s look at the memory footprint of an application, on the right here.

The first problem is that only **some** data are safe to approximate, these are marked here as blue. MemSZ leaves non-approximable data entirely uncompressed.

The second problem is that lossy compression may, over time, accumulate enough errors to become infeasible. MemSZ disables compressions in such cases.

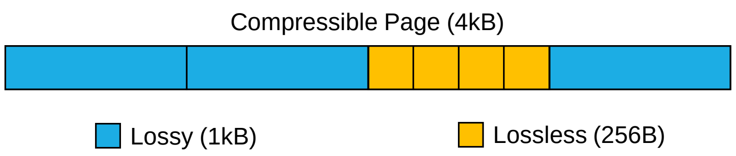
Lossless compression is a complement to this. While it is safely applicable to all data, its compression ratio is limited.

L^{2}C combines lossy and lossless compression to achieve the benefits of both. Lossy compression gives good compression ratios on approximable data. Lossless compression is applied to the rest of the footprint. When lossy compression fails, the offending block falls back to lossless compression.



Block Size

- Expected maximum compression ratio
  - Memory transfer smaller than 1CL is wasteful
  - Compress a block to 1CL, no further
- Lossy aiming for 16x
  - 16 CL block size (1kB)
- Lossless up to 4x
  - 4 CL block size (256B)



32 2022-02-19

As I mentioned earlier, there is no bandwidth gain from compressing a block to be smaller than a single cache line.

Conversely, compressing too much data together may lead to overfetching, since a larger block may not see sufficient reuse.

As a result, the optimal block size is such that the maximum expected compression brings the block to a size of one cache line.

The lossy compressor is designed for up to 16x compression, which is why it operates on blocks of 16 cache lines

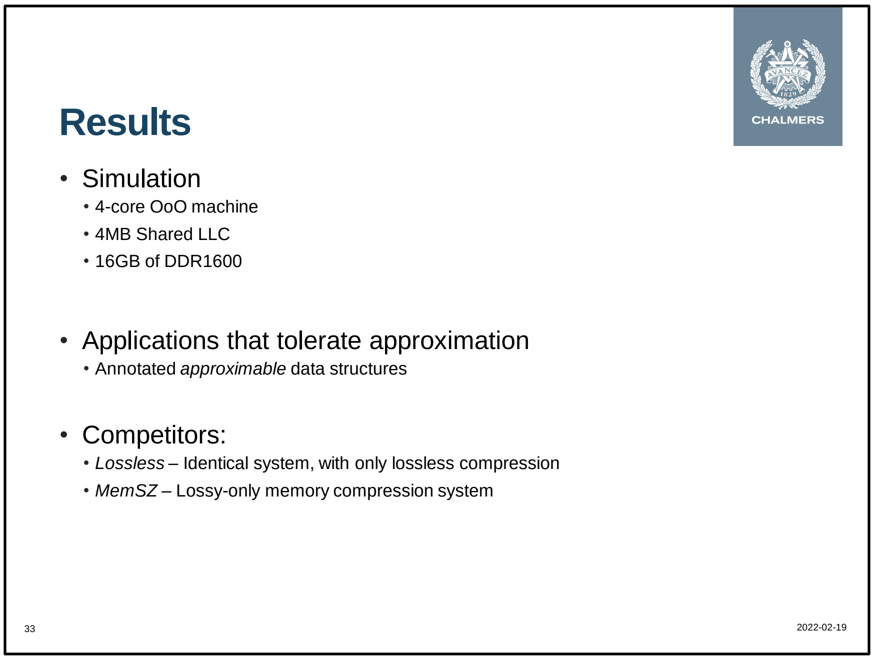
Lossless compression typically achieves up to 4x compression, so the optimal block size is 4 cache lines.

In order to be able to switch between lossy and lossless, the memory layout must support both of these sizes, interchangeably.

L2C divides 4kB pages into 1kB portions, and each such portion uses one compression method.

For example, if a page is approximable, it may start out with all portions compressed lossily. At some point, error accumulates and a 1kB block must fall back to lossless. The 1kB lossy block becomes four smaller lossless blocks.

This way, data can be switched between lossy and lossless compression at runtime, without having to reorganize the rest of the page.

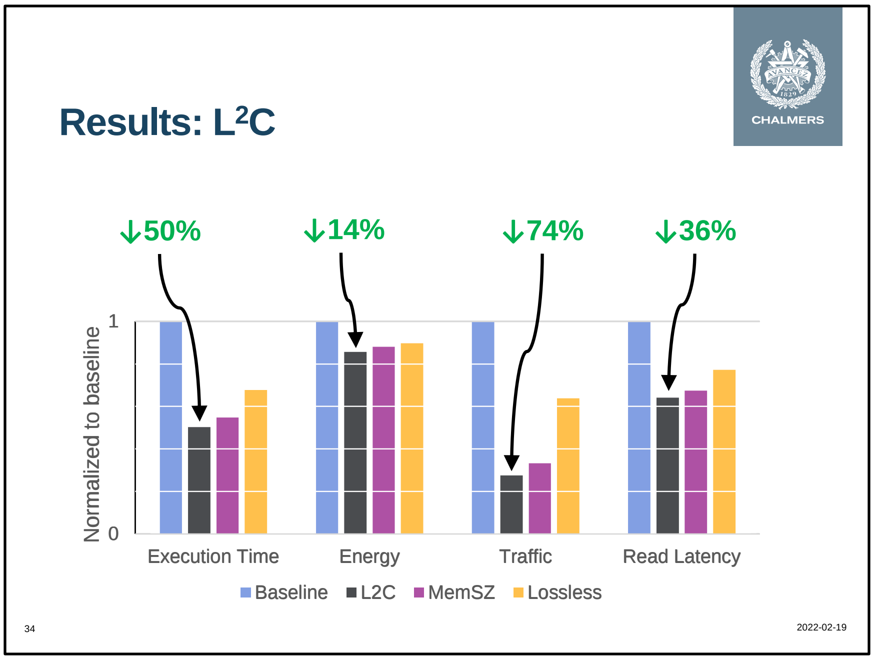


Results

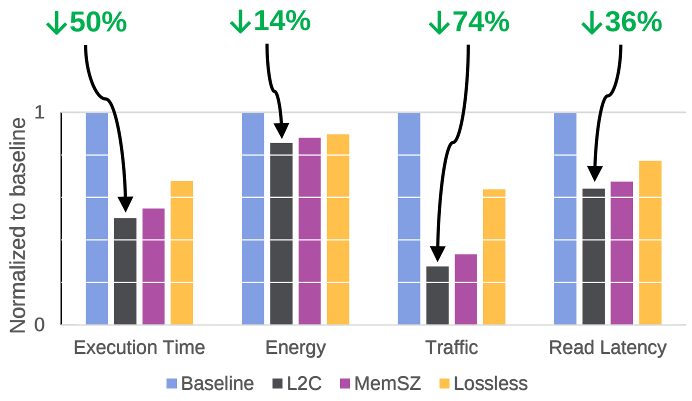
- Simulation
  - 4-core OoO machine
  - 4MB Shared LLC
  - 16GB of DDR1600
- Applications that tolerate approximation
  - Annotated *approximable* data structures
- Competitors:
  - *Lossless* – Identical system, with only lossless compression
  - *MemSZ* – Lossy-only memory compression system

33 2022-02-19

To evaluate L^{2}C, I've once again run simulations with a suite of approximation-tolerant benchmarks. To see the effects of combined compression, I compare it to two very similar systems – one which uses only lossless compression and MemSZ which is lossy-only.



Results: L^{2}C



34

2022-02-19

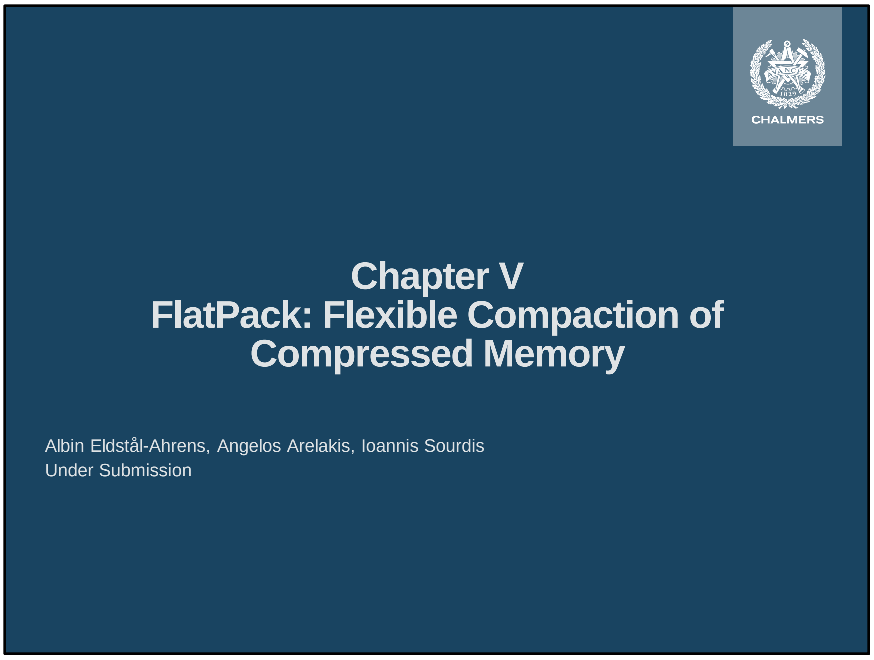
As expected, the combination of two mostly independent compression approaches leads to benefits outstripping either one.

L2C improves performance by a mean 50%.

Energy consumption is reduced by 14% on average.

Memory traffic sees an average reduction of 74%.

As a result, average memory access time is reduced by 36%.

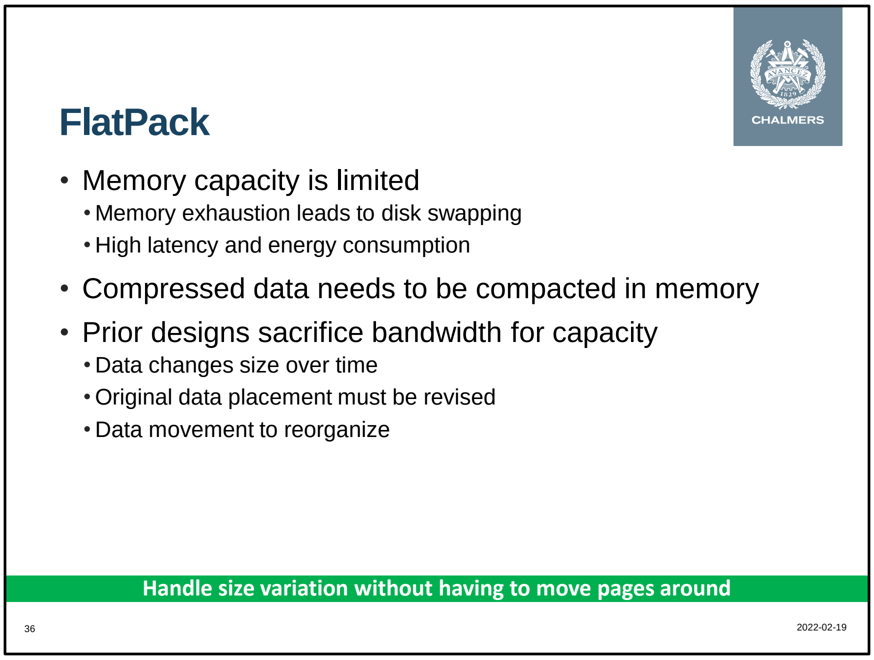


Chapter V
FlatPack: Flexible Compaction of Compressed Memory

Albin Eldstål-Ahrens, Angelos Arelakis, Ioannis Sourdis
Under Submission

The final piece of the puzzle is presented in Chapter 5, a memory compaction system named FlatPack.

This work is currently under submission.



FlatPack

- Memory capacity is limited
  - Memory exhaustion leads to disk swapping
  - High latency and energy consumption
- Compressed data needs to be compacted in memory
- Prior designs sacrifice bandwidth for capacity
  - Data changes size over time
  - Original data placement must be revised
  - Data movement to reorganize

Handle size variation without having to move pages around

36 2022-02-19

As I mentioned back at the beginning, the second big bottleneck of the memory system is **memory capacity**.

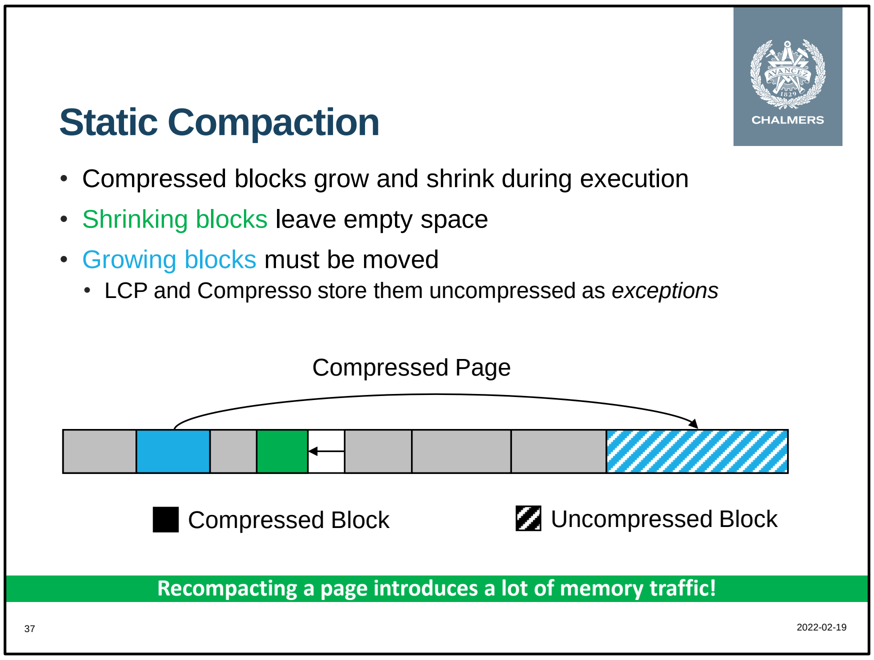
Exhausting main memory leads to *swapping*, where pages have to be moved between the slow main memory and even slower persistent storage.

Compression alone will not improve memory capacity. AVR, MemSZ and L^{2}C all compress blocks without affecting their memory placement, which means that memory capacity remains unchanged.

Prior memory compaction systems target memory capacity, but introduce traffic overheads.

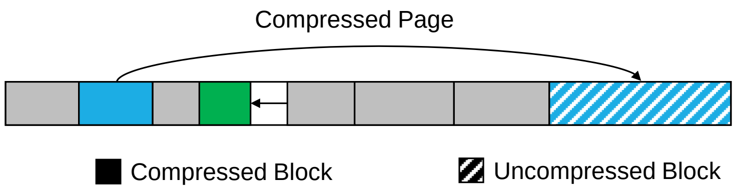
This is mainly because data compressibility varies over time, and thus compressed blocks need to be reorganized.

FlatPack is a memory compaction system which is designed to handle that variation, reorganizing blocks on the fly with minimal traffic overhead.



Static Compaction

- Compressed blocks grow and shrink during execution
- Shrinking blocks leave empty space
- Growing blocks must be moved
  - LCP and Compresso store them uncompressed as *exceptions*



Recompacting a page introduces a lot of memory traffic!

37 2022-02-19

The overhead of existing memory compaction systems stems from a static organization of blocks.

If we compress a page full of data blocks and pack them densely in main memory, we get the best possible compaction **for the time being**.

However, when data are updated during execution, compressibility changes as well!

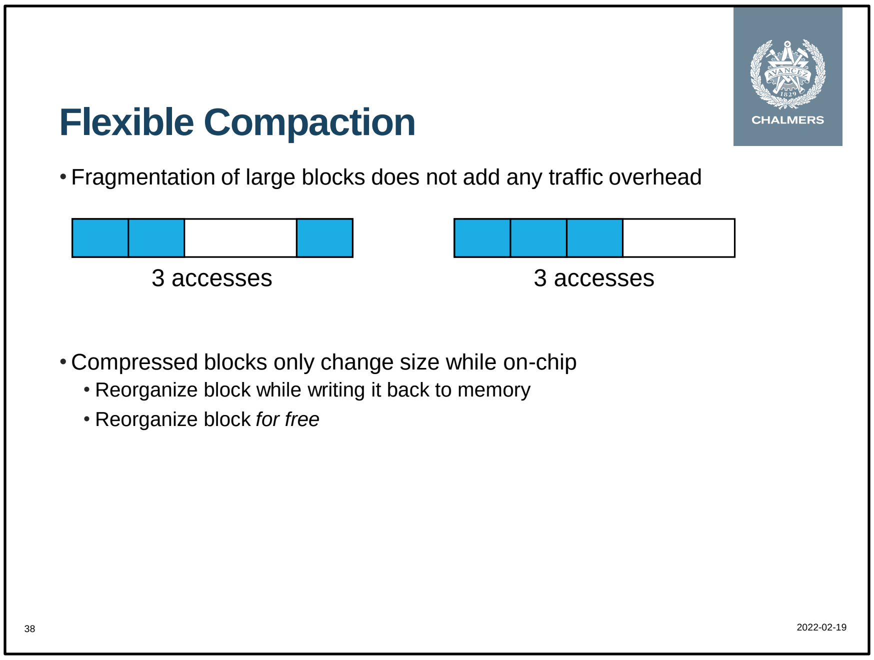
This has two consequences.

If a block, like the green one here, shrinks, it can still fit in the space it was given. It will, however, leave some empty space. The compaction of the page doesn't improve, even though compressibility just went up.

If a block, like the blue one, grows, it won't fit in its original location anymore. State of the art systems handle this by moving the block to a special *exception* space, and leaving it uncompressed.

If this happens to multiple blocks, the exception space will run out at some point. Then, there's no choice but to read the entire page from memory and recompact it.

Such a *page migration* introduces a large traffic overhead.



Flexible Compaction

- Fragmentation of large blocks does not add any traffic overhead

3 accesses   3 accesses

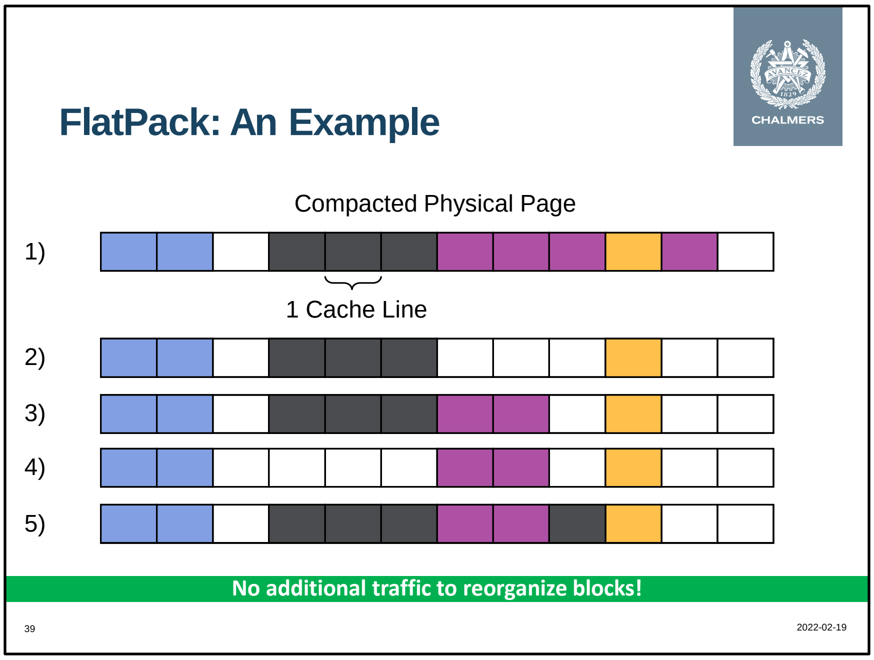
- Compressed blocks only change size while on-chip
  - Reorganize block while writing it back to memory
  - Reorganize block *for free*

38   2022-02-19

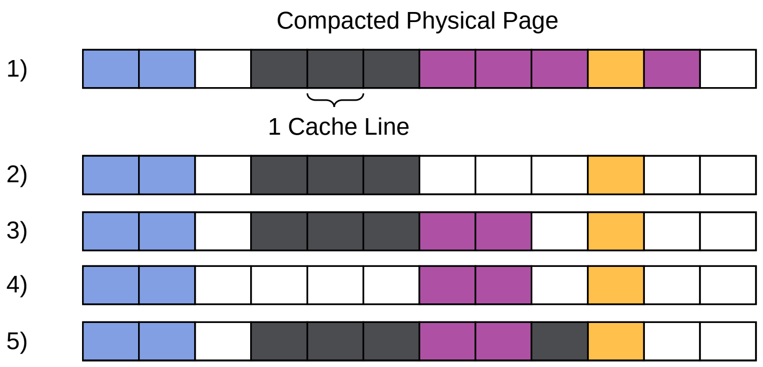
FlatPack is a different approach to memory compaction, which hinges upon two observations.

First, large blocks require multiple accesses to read from memory, so they can be fragmented without overhead.
What I mean by this is that a block which is fragmented and spread out, like this, can be accessed just as easily as one that is stored in contiguous memory, like this.

Second, compressed blocks only change size when all the data is on-chip, since that's where compression takes place.
That means that writing a block back to memory and reorganizing the block in memory can be done at the same time.



FlatPack: An Example



No additional traffic to reorganize blocks!

39 2022-02-19

Let me show you how FlatPack makes use of these properties.

Let's look at a compacted page in physical memory. This has some size smaller than the usual 4kB.

We divide the physical page into *slots*, each the same size as a cache line. This means that one *slot* can be read or written in one memory transfer.

Let's populate this page with compressed data. As we've discussed, compressed blocks vary in size.

To place these blocks in memory, we fragment them at the same granularity as the slots.

(1) We can then place these fragments in memory, here's an example of a valid placement.

Now, let's say that the purple block is read to chip and the application updates the data. When the block is written back to memory, it must first be compressed.

As it turns out, the compressed size is now smaller than it was before. The block shrank.

(2) FlatPack invalidates the old placement of this block, without actually writing to memory.

(3) Then, a new set of slots are selected, and the block is written back to memory. As you can see, two slots in memory were freed up. We've *reorganized* the purple block as part of writing it back to memory.

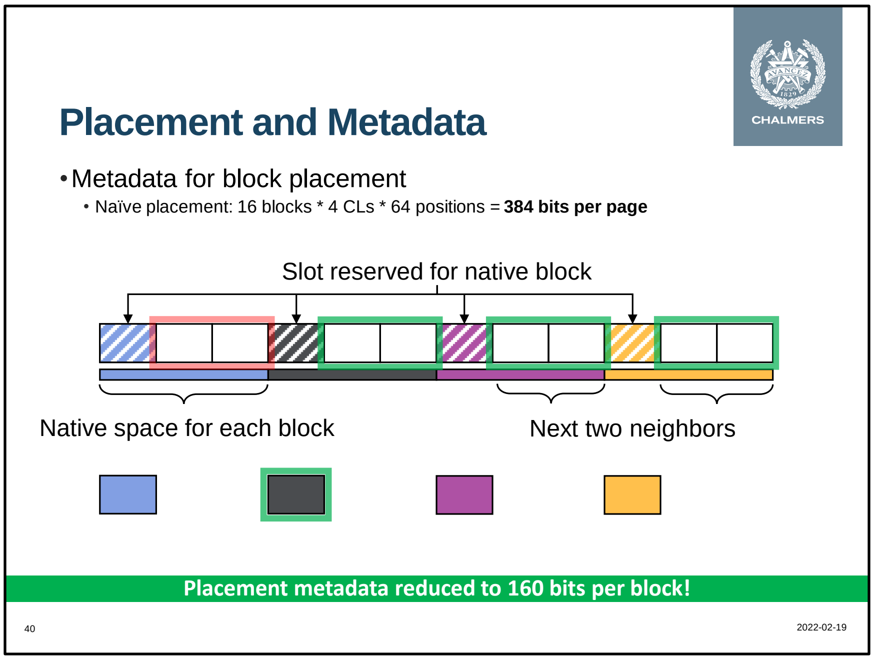
The big feature here, is that these newly freed slots can be reused for other data.

Let’s say the gray block is read to chip, and again has its data updated by the application. This time, the block grew slightly. FlatPack will, again, invalidate the existing placement of the block (4) and find a suitable set of slots before writing the data back to memory (5).

As you can see, this has two big benefits over the static designs I described earlier.

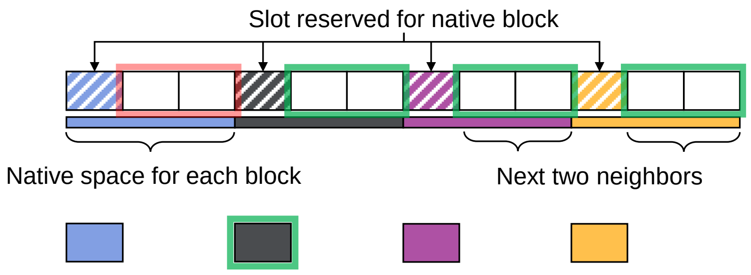
First, no block has had to revert to uncompressed storage.

Second, we’ve incurred only the traffic of reading and writing the blocks, exactly as we would have without compaction. No additional traffic overhead is caused by the reorganization of the blocks.



Placement and Metadata

- Metadata for block placement
  - Naïve placement: 16 blocks * 4 CLs * 64 positions = **384 bits per page**



Placement metadata reduced to 160 bits per block!

40 2022-02-19

This sounds great! Why doesn’t every system just place blocks wherever they fit?

The limitation is metadata – the additional information used to encode where things are stored and how. If every cache line in a page could be freely placed, the metadata for placement alone would be up to 48 bytes per page.

To reduce this overhead, FlatPack introduces a few optimizations and limitations.

Let’s look at a page again, with four blocks.

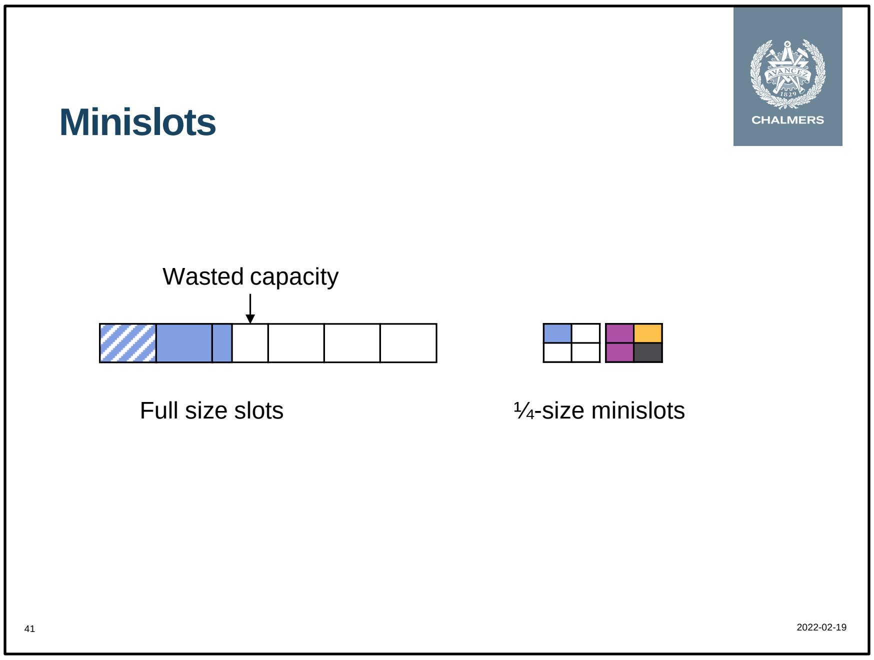
FlatPack divides the physical page so that each block has an equally sized **native** space.

Since every block contains at least *some* data, it follows that it needs at least one slot. Therefore, FlatPack reserves one slot for each block and needs no metadata to know what is stored there.

Additional parts of a block are allowed to go in the block’s own native space. It is also allowed to make use of the native spaces of its two following neighbor blocks. The gray block, in this example, may use space native to purple and yellow.

A block cannot be placed in any other free slots, except for these.

These limitations bring the placement metadata down to 20 bytes per page.



Minislots

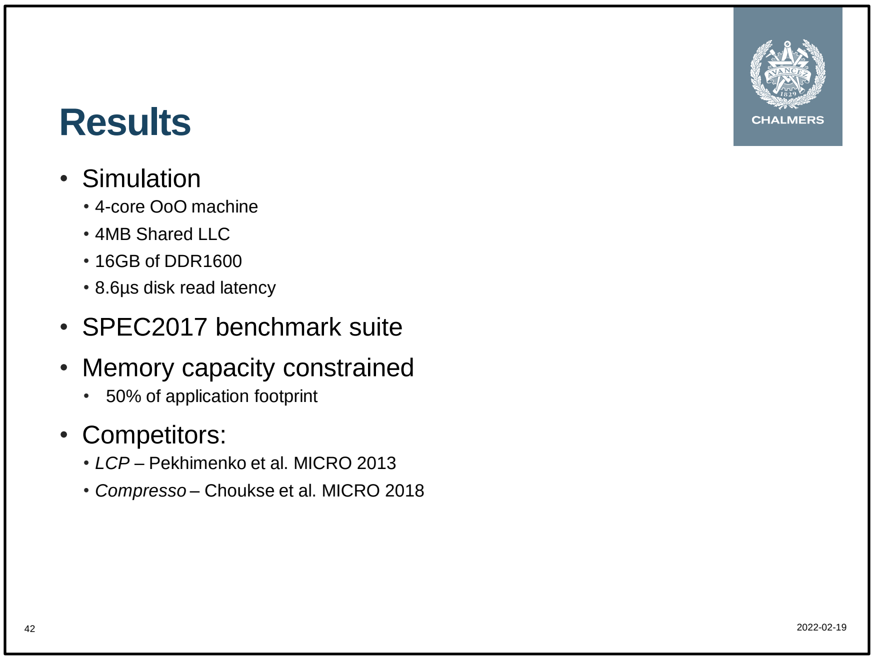
41 2022-02-19

One potential problem with large blocks and fragmentation at the cache line granularity is rounding.

It's unlikely that a block will be compressed to exactly fit within its last slot, so there will be some capacity wasted.

In order to mitigate this, FlatPack designates a part of each physical page for use as **minislots**, which are one quarter the size.

Minislots are assigned to blocks according to the same rules as regular slots, and allow the leftover data of multiple blocks to be packed together more densely.



Results

- Simulation
  - 4-core OoO machine
  - 4MB Shared LLC
  - 16GB of DDR1600
  - 8.6µs disk read latency
- SPEC2017 benchmark suite
- Memory capacity constrained
  - 50% of application footprint
- Competitors:
  - *LCP* – Pekhimenko et al. MICRO 2013
  - *Compresso* – Choukse et al. MICRO 2018

42 2022-02-19

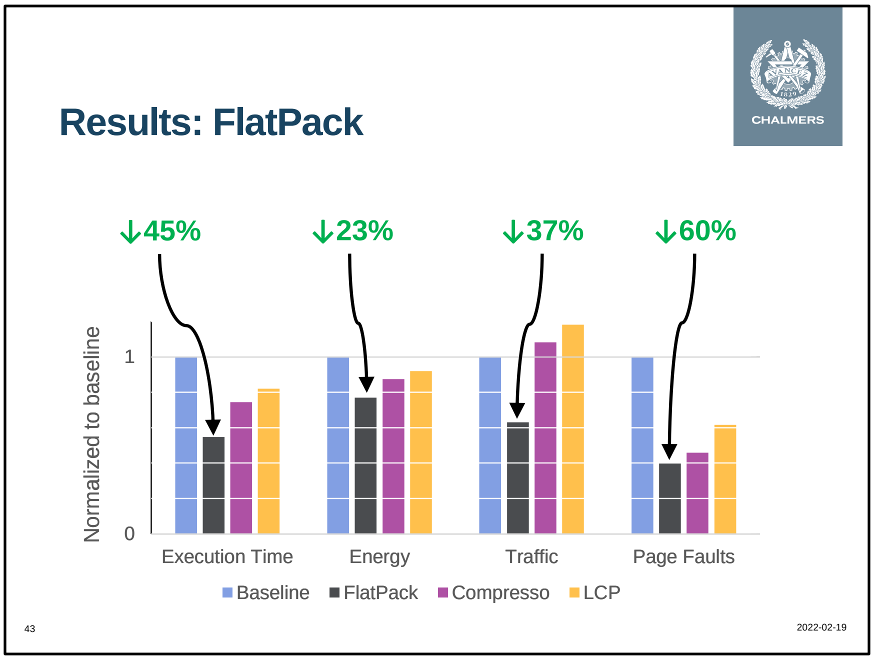
I've evaluated FlatPack in simulation, using the SPEC2017 benchmark suite.

To see the effect on page faults, each experiment simulates a mostly full main memory. 50% of the baseline's active footprint is made available.

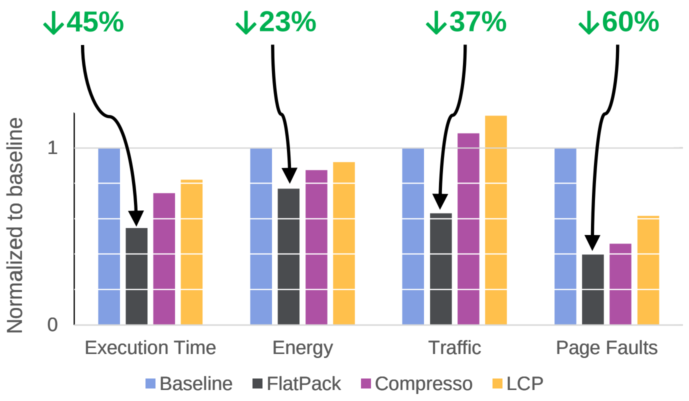
I compare FlatPack to two prior system designs.

LCP uses a static compaction method, where all cache lines within the same page are allocated identical compressed space.
Compresso allows the sizes to differ from one another, but still packs them statically.

All three designs are equipped with identical lossless compressors.



Results: FlatPack



43 2022-02-19

FlatPack is able to reduce execution time by 45%.

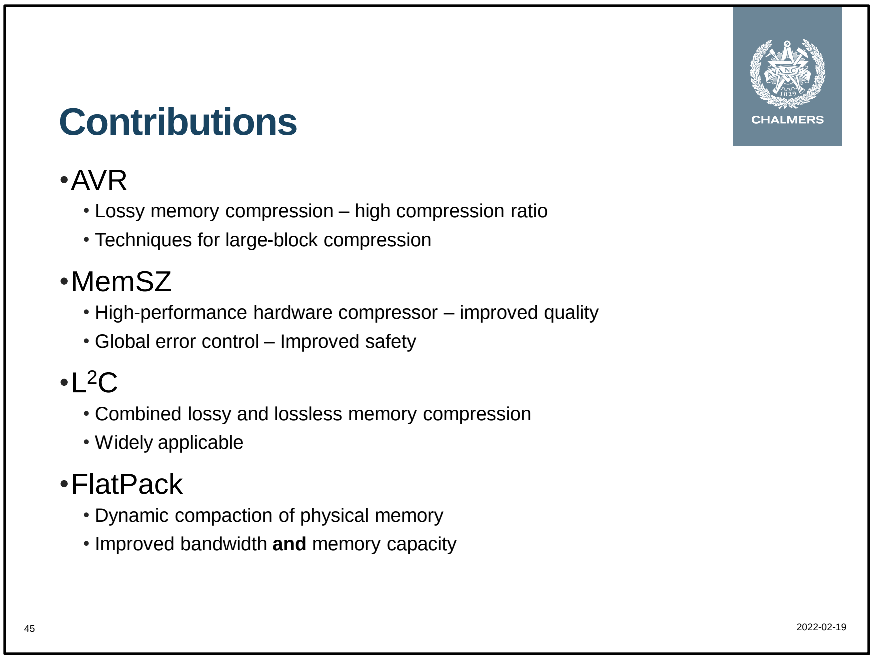
System energy consumption is reduced by 23%, thanks to the performance increase and reduced memory activity.

The main claim of FlatPack is that it saves on memory traffic, compared to existing designs. In these experiments, both Compresso an LCP had net traffic overheads, while FlatPack gave a reduction of 37% on average.

Finally, the main purpose of memory compaction is to reduce the number of *page faults*, where pages are moved between memory and disk. All three designs achieve this, and FlatPack reduces the baseline’s page fault count by 60%.

Conclusion

That concludes the last of the contribution chapters, and leads us to some concluding remarks.



Contributions

- AVR
  - Lossy memory compression – high compression ratio
  - Techniques for large-block compression
- MemSZ
  - High-performance hardware compressor – improved quality
  - Global error control – Improved safety
- L^{2}C
  - Combined lossy and lossless memory compression
  - Widely applicable
- FlatPack
  - Dynamic compaction of physical memory
  - Improved bandwidth **and** memory capacity

45

2022-02-19

In summary, this thesis presents four different hardware systems, aimed at mitigating the bandwidth and capacity bottlenecks in the memory hierarchy.

AVR sets out a basic architecture for lossy compression of large blocks, and an LLC design to support it.

MemSZ expands on this with a more advanced compression scheme and a persistent global error limiting mechanism

L2C combines this lossy compression with lossless, both as a fallback and to be applicable across the entire memory footprint.

Finally, FlatPack is a memory compaction system which leverages the large compression blocks to mitigate the bandwidth overheads of compaction. It provides the capacity benefits of memory compaction and retains the bandwidth improvements of compression.