Compress Objects, Not Cache Lines: An Object-Based Compressed Memory Hierarchy

Po-An Tsai and Daniel Sanchez
Prior memory compression techniques are limited to compressing *cache lines*.
Prior memory compression techniques are limited to compressing cache lines

- Data movement limits performance and efficiency
  - A memory access takes $100X$ the latency and $1000X$ the energy of a FP operation
Data movement limits performance and efficiency
- A memory access takes $100X$ the latency and $1000X$ the energy of a FP operation

Applying hardware-based compression to the memory hierarchy to reduce data movement thus becomes beneficial

Prior memory compression techniques are limited to compressing cache lines
Prior memory compression techniques are limited to compressing cache lines

- Data movement limits performance and efficiency
  - A memory access takes $100X$ the latency and $1000X$ the energy of a FP operation

- Applying hardware-based compression to the memory hierarchy to reduce data movement thus becomes beneficial

To support random accesses, the memory hierarchy transfers cache lines between levels

→ Prior techniques are thus limited to compressing cache lines
Challenges due to compressing at cache-line granularity
Challenges due to compressing at cache-line granularity

1. Locating the compressed cache line (architecture)

   Fixed-size cache lines become variable-size compressed blocks
   \[\rightarrow\] HW needs to translate uncompressed addresses to compressed blocks
Challenges due to compressing at cache-line granularity

1. Locating the compressed cache line (architecture)
   - Fixed-size cache lines become variable-size compressed blocks
   - HW needs to translate uncompressed addresses to compressed blocks

2. Compressing cache lines (algorithm)
   - Cache lines are small, and decompression latency is on the critical path
   - HW cannot compress more than 64B at a time
   - Only low-latency algorithms are practical
Prior compressed memory architectures sacrifice compression ratio for low latency
Prior compressed memory architectures sacrifice compression ratio for low latency

- They aim to quickly **translate** uncompressed to compressed addresses
  - Example: Linearly compressed pages

- **Example**: Linearly compressed pages

![Diagram](image)
Prior compressed memory architectures sacrifice compression ratio for low latency

- They aim to quickly **translate** uncompressed to compressed addresses

  - Example: Linearly compressed pages

    - Original cache line address
    - Compressed block address
    - Uncompressed format

    - 4KB page
    - 64B lines
Prior compressed memory architectures sacrifice compression ratio for low latency

- They aim to quickly **translate** uncompressed to compressed addresses

- Example: Linearly compressed pages

  LCP compresses page by page to leverage VM for translation
  → Fast and low overhead

  LCP forces cache lines in the same page to compress into the same size
  → Sacrifice compression ratio
Prior compressed memory architectures sacrifice compression ratio for low latency

- They aim to quickly **translate** uncompressed to compressed addresses
  - Example: Linearly compressed pages

- LCP compresses page by page to leverage VM for translation
  - Fast and low overhead

- LCP forces cache lines in the same page to compress into the same size
  - Sacrifice compression ratio

- Other techniques make similar tradeoffs
  - E.g., 4 different sizes for cache lines in a page

[DMC, Kim et al., PACT'17]
[Compresso, Choukse et al., MICRO'18]

[RMC, Ekman and Stenstorm, HPCA'06]
Prior compression algorithms are limited to exploit redundancy within a cache line to achieve low latency.
Prior compression algorithms are limited to exploit redundancy within a cache line to achieve low latency.

Example: Base-Delta-Immediate compression

<table>
<thead>
<tr>
<th>Uncompressed layout</th>
<th><strong>Int</strong> array</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
<td>100</td>
<td>102</td>
<td>101</td>
<td>103</td>
<td>103</td>
<td>102</td>
<td>104</td>
<td>108</td>
<td>109</td>
<td>109</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Float</strong> array</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>......</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Reference</strong> array</th>
<th>0x18</th>
<th>0x30</th>
<th>0x48</th>
<th>......</th>
</tr>
</thead>
</table>

Example: Base-Delta-Immediate, Pekhimenko et al., PACT'12
Prior compression algorithms are limited to exploit redundancy within a cache line to achieve low latency.

Example: Base-Delta-Immediate compression

Uncompressed layout

<table>
<thead>
<tr>
<th>Int array</th>
<th>100</th>
<th>100</th>
<th>102</th>
<th>101</th>
<th>103</th>
<th>103</th>
<th>102</th>
<th>104</th>
<th>108</th>
<th>109</th>
<th>109</th>
<th>111</th>
</tr>
</thead>
</table>

Compressed layout

<table>
<thead>
<tr>
<th>Int array</th>
<th>100</th>
<th>+</th>
<th>+</th>
<th>+</th>
<th>+</th>
<th>+</th>
<th>+</th>
<th>108</th>
<th>+</th>
<th>+</th>
<th>+</th>
<th>+</th>
</tr>
</thead>
</table>

64B cache line

Work well on arrays: Homogeneous, regular

[FP-H, Arelakis et al., MICRO’15] [BPC, Kim et al., ISCA’16]
Prior compression algorithms are limited to exploit redundancy within a cache line to achieve low latency.

Example: Base-Delta-Immediate compression

<table>
<thead>
<tr>
<th>Uncompressed layout</th>
<th>Compressed layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int array</td>
<td></td>
</tr>
<tr>
<td>100 100 102 101 103 103 102 104 108 109 109 111</td>
<td>100 + + + + + + + 108 + + +</td>
</tr>
<tr>
<td>Float array</td>
<td></td>
</tr>
<tr>
<td>1.1 1.2 1.3</td>
<td>1.1 1.2 1.3</td>
</tr>
<tr>
<td>Reference array</td>
<td></td>
</tr>
<tr>
<td>0x18 0x30 0x48</td>
<td>0x18 0x30 0x48</td>
</tr>
</tbody>
</table>

Work well on arrays: Homogeneous, regular

<table>
<thead>
<tr>
<th>COMPRESSION RATIO</th>
<th>FFT</th>
<th>SPMV</th>
</tr>
</thead>
<tbody>
<tr>
<td>No compression</td>
<td>1.67</td>
<td>1.55</td>
</tr>
<tr>
<td>Prior work</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[FP-H, Arelakis et al., MICRO’15] [BPC, Kim et al., ISCA’16]
Prior compression algorithms work poorly on objects
Prior compression algorithms work poorly on objects

Work poorly on objects: Heterogeneous, irregular

Object A1  Object B  Object A2  Object C
Prior compression algorithms work poorly on objects

Work poorly on objects: Heterogeneous, irregular

64B cache line

Object A1

Object B

Object A2

Object C

Little redundancy within a cache line
Prior compression algorithms work poorly on objects

Work poorly on objects: Heterogeneous, irregular

Object A1 100 1.1 0x18
Object B 102 1.3 0x48
Object C

Little redundancy within a cache line

64B cache line

Array-heavy apps: 61% compression ratio

Object-heavy apps: 14% compression ratio
Objects, not cache lines, are the natural unit of compression.
Objects, not cache lines, are the natural unit of compression

Insight 1:
Object-based applications always follow pointers to access objects
Objects, not cache lines, are the natural unit of compression

Insight 1:
Object-based applications always follow pointers to access objects

Uncompressed layout

Object A1  Object B1  Object A2  Object C  Object B2

0x00 0xFF
Objects, not cache lines, are the natural unit of compression

**Insight 1:**
Object-based applications always follow pointers to access objects

Uncompressed layout

<table>
<thead>
<tr>
<th>Object A1</th>
<th>Object B1</th>
<th>Object A2</th>
<th>Object C</th>
<th>Object B2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Idea 1:**
Point directly to the location of compressed objects to avoid uncompressed-to-compressed address translation!

Compressed layout

<table>
<thead>
<tr>
<th>Object A1</th>
<th>Object B1</th>
<th>Object A2</th>
<th>Object C</th>
<th>Object B2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td></td>
<td></td>
<td>0xDF</td>
<td>0xFF</td>
</tr>
</tbody>
</table>
Objects, not cache lines, are the natural unit of compression.
Objects, not cache lines, are the natural unit of compression

Insight 2:
There is significant redundancy across objects of the same type
Objects, not cache lines, are the natural unit of compression

**Insight 2:**
There is significant redundancy across objects of the same type
Objects, not cache lines, are the natural unit of compression

Insight 2:
There is significant redundancy across objects of the same type

Compressed layout

Object A1  Object B1  Object A2  Object C  Object B2

Compressed layout

Further compressed layout

\[ \Delta A1 = \text{Bytes that differ from a shared base object} \]
Compressing objects would be hard to do on cache hierarchies
Compressing objects would be hard to do on cache hierarchies

- Ideally, we want a memory system that
  - Moves objects, rather than cache lines
  - Transparently updates pointers during compression
Compressing objects would be hard to do on cache hierarchies

- Ideally, we want a memory system that
  - Moves objects, rather than cache lines
  - Transparently updates pointers during compression

- Therefore, we realize our ideas on *Hotpads*
  - A recent *object-based* memory hierarchy
Baseline system: Hotpads overview
Baseline system: Hotpads overview
Baseline system: Hotpads overview

- Data array
  - Managed as a circular buffer using simple sequential allocation
  - Stores variable-sized objects compactly
Baseline system: Hotpads overview

- Data array
  - Managed as a circular buffer using simple sequential allocation
  - Stores variable-sized objects compactly
Baseline system: Hotpads overview

- Data array
  - Managed as a circular buffer using simple sequential allocation
  - Stores variable-sized objects compactly
Baseline system: Hotpads overview

- Data array
  - Managed as a circular buffer using simple sequential allocation
  - Stores variable-sized objects compactly
  - Can store variable-sized compressed objects compactly too!
Baseline system: Hotpads overview

- **Data array**
  - Managed as a circular buffer using simple sequential allocation
  - Stores variable-sized objects compactly
  - Can store variable-sized *compressed* objects compactly too!

- **C-Tags**
  - Decoupled tag store

- **Metadata**
  - Pointer? valid? dirty? recently-used?
Hotpads moves objects instead of cache lines
Hotpads moves objects instead of cache lines

Example object:
```java
class ListNode {
    int value;
    ListNode next;
}
```

Initial state.
Hotpads moves objects instead of cache lines

Example object:

```java
class ListNode {
    int value;
    ListNode next;
}
```

Program code:

```java
int v = A.value;
```

Initial state.

A copied into L1 pad.
Hotpads moves objects instead of cache lines

Example object:

```java
class ListNode {
    int value;
    ListNode next;
}
```

Program code:

1. `int v = A.value;`  
   - A copied into L1 pad.

2. `v = A.next.value;`  
   - B copied into L1 pad.
**Hotpads moves objects instead of cache lines**

Example object:

```java
class ListNode {
    int value;
    ListNode next;
}
```

Program code:

1. `int v = A.value;`
   - A copied into L1 pad.
2. `v = A.next.value;`
   - B copied into L1 pad.

Hotpads takes control of the memory layout, hides pointers from software, and encodes object information in pointers.

Fetching `size` words from the starting address yields the entire object.
Hotpads moves objects instead of cache lines

Example object:

```cpp
class ListNode {
    int value;
    ListNode next;
}
```

Program code:

1. `int v = A.value;`
2. `v = A.next.value;`

Hotpads takes control of the memory layout, hides pointers from software, and encodes object information in pointers.

Compressed size

Fetching **compressed** size words from the starting **compressed** address yields the entire **compressed** object.
Hotpads updates pointers among objects on evictions
Hotpads updates pointers among objects on evictions

L1 pad is full because of fetched objects or newly-allocate objects

L1 pad is now full, triggering a bulk eviction in HW.
Hotpads updates pointers among objects on evictions

3. L1 pad is full because of fetched objects or newly-allocate objects

After an L1 bulk eviction:
Pointers are updated to point to the new locations.

4. L1 pad is now full, triggering a bulk eviction in HW.

After an L1 bulk eviction:
Pointers are updated to point to the new locations.

Copied objects (A) are back to old location
New objects (D) are sequentially allocated

Free space
Hotpads updates pointers among objects on evictions

- **L1 pad is full** because of fetched objects or newly-allocate objects
- **After an L1 bulk eviction:**
  - Pointers are updated to point to the new locations.
  - Copied objects (A) are back to old location
  - New objects (D) are sequentially allocated

- Bulk eviction amortizes the cost of finding and updating pointers across objects
Hotpads updates pointers among objects on evictions

3. L1 pad is full because of fetched objects or newly-allocate objects

4. After an L1 bulk eviction:
   Pointers are updated to point to the new locations.

- Bulk eviction amortizes the cost of finding and updating pointers across objects
- Since updating pointers already happens in Hotpads, there is no extra cost to update them to compressed locations!
Zippads: Locating objects without translations
Zippads: Locating objects without translations

- Zippads leverages Hotpads to
  - Manipulate and compress objects rather than cache lines
  - Avoid translation by pointing directly to compressed objects during evictions
Zippads: Locating objects without translations

- Zippads leverages Hotpads to
  - Manipulate and compress objects rather than cache lines
  - Avoid translation by pointing directly to compressed objects during evictions
Zippads: Locating objects without translations

- Zippads leverages Hotpads to
  - Manipulate and compress objects rather than cache lines
  - Avoid translation by pointing directly to compressed objects during evictions

![Diagram of Zippads](image)
Zippads compresses objects when they move
Zippads compresses objects when they move

- Objects are compressed during bulk object evictions
Zippads compresses objects when they move

- Objects are compressed during bulk object evictions

**Case 1: Newly moved objects**

- Object (uncompressed)
- L2 pad

Objects start their lifetime uncompressed in private levels
Objects are compressed during bulk object evictions

**Case 1: Newly moved objects**

Objects start their lifetime uncompressed in private levels.

When objects are evicted into a compressed level, they are compressed in that level and store compactly.
Zippads compresses objects when they move

- Objects are compressed during bulk object evictions

**Case 1: Newly moved objects**

- Object (uncompressed)
- L2 pad
- Compression HW
- L3 pad

Objects start their lifetime uncompressed in private levels.

When objects are evicted into a compressed level, they are compressed in that level and store compactly.

**Piggyback the bulk eviction process to find and update all pointers at once, amortizing update costs**
Zippads compresses objects when they move.

- Objects are compressed during bulk object evictions.
Zippads compresses objects when they move

- Objects are compressed during bulk object evictions

**Case 2: Dirty writeback**

- **Updated object (uncompressed)**
  - **L2 pad**
  - **Compression HW**
  - **Old object (compressed)**
  - **L3 pad**
  - **Objects**
  - **Free space**
Zippads compresses objects when they move

- Objects are compressed during bulk object evictions

**Case 2: Dirty writeback**

- **L2 pad**: Updated object (uncompressed)
- **Compression HW**
- **L3 pad**: Old object (compressed)
  - Objects
  - Free space

If new size $\leq$ old size
Zippads compresses objects when they move

- Objects are compressed during bulk object evictions

**Case 2: Dirty writeback**

- Updated object (uncompressed)
- L2 pad
- Compression HW
- L3 pad
  - Objects
  - Old object (compressed)
  - Objects
  - Free space

If new size <= old size

If new size > old size

- Objects
- Updated object (compressed)
- Unused space
- Objects
- Free space

- Objects
- Forwarding thunk
- Unused space
- Objects
- Updated object (compressed)
Zippads compresses objects when they move

- Objects are compressed during bulk object evictions

**Case 2: Dirty writeback**

Periodic compaction reclaims those unused spaces (Bulk eviction in on-chip pads, GC in main memory)
Zippads uses pointers to accelerate decompression
Zippads uses pointers to accelerate decompression

- **Every object access starts with a pointer!**
  - Pointers are updated to the compressed locations, so no translation is needed
Zippads uses pointers to accelerate decompression

- Every object access starts with a pointer!
  - Pointers are updated to the compressed locations, so no translation is needed

- Prior work shows it’s beneficial to use different algorithms for various patterns
  - Zippads encodes compression metadata in pointers to decompress objects quickly

![Compression Encoding Diagram]
Zippads uses pointers to accelerate decompression

- Every object access starts with a pointer!
  - Pointers are updated to the compressed locations, so no translation is needed

- Prior work shows it’s beneficial to use different algorithms for various patterns
  - Zippads encodes compression metadata in pointers to decompress objects quickly

- Zippads thus knows how to locate and what decompression algorithm to use when accessing compressed objects with pointers
COCO: Cross-object-compression algorithm
COCO: Cross-object-compression algorithm

- COCO exploits similarity across objects with shared **base objects**
  - A collection of representative objects
COCO: Cross-object-compression algorithm

- COCO exploits similarity across objects with shared **base objects**
  - A collection of representative objects
COCO: Cross-object-compression algorithm

- COCO exploits similarity across objects with shared **base objects**
  - A collection of representative objects

![Diagram of COCO algorithm](image)

- Uncompressed object
- Compression HW
- Base object
- Compressed object
  - Pointer to the base object
  - Bytes that are different
COCO: Cross-object-compression algorithm

- COCO requires accessing base objects for every compression/decompression
COCO: Cross-object-compression algorithm

- COCO requires accessing base objects for every compression/decompression.

- Caching base objects avoids extra latency and bandwidth to fetch them.

- A small (8KB) base object cache works well.
  - Few types account for most accesses.
See paper for additional features and details

- Compressing large objects with subobjects and allocate-on-access
- COCO compression/decompression circuit RTL implementation details
- Details on integrating Zippads and COCO
- Discussion on using COCO with conventional memory hierarchies
Evaluation
Evaluation

- We simulate Zippads using MaxSim [Rodchenko et al., ISPASS’17]
  - A simulator combining ZSim and Maxine JVM
Evaluation

- We simulate Zippads using MaxSim [Rodchenko et al., ISPASS’17]
  - A simulator combining ZSim and Maxine JVM

- We compare 4 schemes
Evaluation

- We simulate Zippads using MaxSim [Rodchenko et al., ISPASS’17]
  - A simulator combining ZSim and Maxine JVM

- We compare 4 schemes
  - **Uncomp**: Conventional 3-level cache hierarchy with no compression
Evaluation

- We simulate Zippads using MaxSim [Rodchenko et al., ISPASS’17]
  - A simulator combining ZSim and Maxine JVM

- We compare 4 schemes
  - **Uncomp**: Conventional 3-level cache hierarchy with no compression
  - **CMH**: Compressed memory hierarchy
    - LLC: VSC [Alameldeen and Wood, ISCA'04]
    - Main memory: LCP [Pekhimenko et al., MICRO'13]
    - Algorithm: HyComp-style hybrid algorithm
      - BDI + FPC [Arelakis et al., MICRO'15]
Evaluation

- We simulate Zippads using MaxSim [Rodchenko et al., ISPASS’17]
  - A simulator combining ZSim and Maxine JVM

- We compare 4 schemes
  - **Uncomp**: Conventional 3-level cache hierarchy with no compression
  - **CMH**: Compressed memory hierarchy
    - LLC: VSC [Alameldeen and Wood, ISCA’04]
    - Main memory: LCP [Pekhimenko et al., MICRO’13]
    - Algorithm: HyComp-style hybrid algorithm
      - BDI + FPC [Arelakis et al., MICRO’15]
  - **Hotpads**: The baseline system we build on
Evaluation

- We simulate Zippads using MaxSim [Rodchenko et al., ISPASS’17]
  - A simulator combining ZSim and Maxine JVM

- We compare 4 schemes
  - **Uncomp:** Conventional 3-level cache hierarchy with no compression
  - **CMH:** Compressed memory hierarchy
    - LLC: VSC [Alameldeen and Wood, ISCA’04]
    - Main memory: LCP [Pekhimenko et al., MICRO’13]
    - Algorithm: HyComp-style hybrid algorithm
      - BDI + FPC [Arelakis et al., MICRO’15]
  - **Hotpads:** The baseline system we build on
  - **Zippads:** With and without COCO
Evaluation

- We simulate Zippads using MaxSim [Rodchenko et al., ISPASS’17]
  - A simulator combining ZSim and Maxine JVM

- We compare 4 schemes
  - **Uncomp**: Conventional 3-level cache hierarchy with no compression
  - **CMH**: Compressed memory hierarchy
    - LLC: VSC [Alameldeen and Wood, ISCA’04]
    - Main memory: LCP [Pekhimenko et al., MICRO’13]
    - Algorithm: HyComp-style hybrid algorithm
      - BDI + FPC [Arelakis et al., MICRO’15]
  - **Hotpads**: The baseline system we build on
  - **Zippads**: With and without COCO

- Workloads: 8 Java apps with large memory footprint from different domains
Zippads improves compression ratio
Zippads improves compression ratio
Zippads improves compression ratio
Zippads improves compression ratio

Same algo as CMH

Uncomp.  CMH  Zippads-BF  Zippads
Zippads improves compression ratio
Zippads improves compression ratio

![Graph showing compression ratio comparison between Uncomp., CMH, Zippads-BF, and Zippads]

- Uncomp.
- CMH
- Zippads-BF
- Zippads

Legend:
- Same algo as CMH
- CMH algo + COCO
Zippads improves compression ratio

Only 24% better than Uncomp.
Zippads improves compression ratio

- Same algo as CMH
- CMH algo + COCO

Only 24% better than Uncomp.

70% better
Zippads improves compression ratio

- Same algo as CMH
- CMH algo + COCO

Uncomp. | CMH | Zippads-BF | Zippads

Comparison across different benchmarks:
- fft: Uncomp. vs. CMH: 70% better
- spmv: Uncomp. vs. CMH: 2X better
- h2: Uncomp. vs. CMH: Only 24% better than Uncomp.
Zippads improves compression ratio

1. Both Zippads and CMH work well in array-heavy apps
Zippads improves compression ratio

1. Both Zippads and CMH work well in array-heavy apps
2. Zippads works much better than CMH in object-heavy apps
Zippads reduces memory traffic and improves performance
Zippads reduces memory traffic and improves performance

![Bar chart showing memory traffic reduction with Zippads](image)

*Lower is better*
Zippads reduces memory traffic and improves performance

1. CMH reduces traffic by 15% with data compression
Zippads reduces memory traffic and improves performance

1. CMH reduces traffic by 15% with data compression

2. Hotpads reduces traffic by 66% with object-based data movement
Zippads reduces memory traffic and improves performance

1. CMH reduces traffic by 15% with data compression

2. Hotpads reduces traffic by 66% with object-based data movement

3. Zippads combines the benefits of both, reducing traffic by 2X (70% less traffic than CMH)
Zippads reduces memory traffic and improves performance

1. CMH reduces traffic by 15% with data compression

2. Hotpads reduces traffic by 66% with object-based data movement

3. Zippads combines the benefits of both, reducing traffic by 2X (70% less traffic than CMH)

Similar trend in performance:
Zippads is 24% faster than CMH;
30% faster than Uncomp.
Zippads also provides benefits on compiled code
Zippads also provides benefits on compiled code

- We study two object-heavy benchmarks written in C/C++
Zippads also provides benefits on compiled code

- We study two object-heavy benchmarks written in C/C++
Zippads also provides benefits on compiled code

- We study two object-heavy benchmarks written in C/C++

[Graph showing compression ratios for gcbench and silo]

Zippads again works much better than CMH in compressing memory footprint
Zippads also provides benefits on compiled code

- We study two object-heavy benchmarks written in C/C++

Zippads again works much better than CMH in compressing memory footprint

Zippads improves both memory traffic and performance the most
See paper for more evaluation results

- Zippads hardware storage overhead analysis
- COCO RTL implementation result
- Comparison against CMH with hardware support for memory management
- Zippads analysis
  - Base object cache size sensitivity study
  - Overflow frequency
We propose the first *object-based* compressed memory hierarchy
We propose the first object-based compressed memory hierarchy

- Prior compressed memory hierarchies focus on compressing cache lines.
  - Require address translation and work poorly on object-heavy apps.
We propose the first object-based compressed memory hierarchy

- Prior compressed memory hierarchies focus on compressing cache lines
  - Require address translation and work poorly on object-heavy apps

- Object-based apps provide new opportunities for compression
  - Always access objects through pointers
  - Have significant redundancy across objects, not within cache lines
We propose the first object-based compressed memory hierarchy

- Prior compressed memory hierarchies focus on compressing cache lines
  - Require address translation and work poorly on object-heavy apps

- Object-based apps provide new opportunities for compression
  - Always access objects through pointers
  - Have significant redundancy across objects, not within cache lines

- We present techniques that compress objects, not cache lines
  - Zippads rewrites pointers to avoid uncompressed-to-compressed address translation
  - COCO compresses across objects to leverage more redundancy
Prior compressed memory hierarchies focus on compressing cache lines
- Require address translation and work poorly on object-heavy apps

Object-based apps provide new opportunities for compression
- Always access objects through pointers
- Have significant redundancy across objects, not within cache lines

We present techniques that compress objects, not cache lines
- Zippads rewrites pointers to avoid uncompressed-to-compressed address translation
- COCO compresses across objects to leverage more redundancy