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

Po-An Tsai
MIT CSAIL
poantsai@csail.mit.edu

Daniel Sanchez
MIT CSAIL
sanchez@csail.mit.edu

Abstract

Existing cache and main memory compression techniques compress data in small fixed-size blocks, typically cache lines. Moreover, they use simple compression algorithms that focus on exploiting redundancy within a block. These techniques work well for scientific programs that are dominated by arrays. However, they are ineffective on object-based programs because objects do not fall neatly into fixed-size blocks and have a more irregular layout.

We present the first compressed memory hierarchy designed for object-based applications. We observe that *(i)* objects, not cache lines, are the natural unit of compression for these programs, as they traverse and operate on object pointers; and *(ii)* though redundancy within each object is limited, redundancy across objects of the same type is plentiful. We exploit these insights through Zippads, an object-based compressed memory hierarchy, and COCO, a cross-object-compression algorithm. Building on a recent object-based memory hierarchy, Zippads transparently compresses variable-sized objects and stores them compactly. As a result, Zippads consistently outperforms a state-of-the-art compressed memory hierarchy: on a mix of array- and object-dominated workloads, Zippads achieves 1.63× higher compression ratio and improves performance by 17%.

CCS Concepts • **Computer systems organization → Processors and memory architectures**.

Keywords cache; memory; object-based; compression.

ACM Reference Format:
Po-An Tsai and Daniel Sanchez. 2019. Compress Objects, Not Cache Lines: An Object-Based Compressed Memory Hierarchy. In *Proceedings of 2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS'19).* ACM, New York, NY, USA, 14 pages. http://dx.doi.org/10.1145/3297858.3304006

ASPLOS'19, April 13–17, 2019, Providence, RI, USA

1 Introduction

Compression has become an attractive technique to improve the performance and efficiency of modern memory hierarchies. Ideally, compressing data at a level of the memory hierarchy (e.g., main memory or the last-level cache) brings two key benefits. First, it increases the effective capacity of that level (e.g., reducing page faults or cache misses). Second, it reduces bandwidth demand to that level, as each access fetches a smaller amount of compressed data. Because accesses to off-chip memory or large on-chip caches are slow and expensive, the benefits of compression justify its overheads. Therefore, prior work has proposed compressed main memory [17, 25, 34, 51] and cache [1, 22, 30, 41, 42] architectures, as well as specialized compression algorithms [4, 5, 24, 35].

Unfortunately, hardware memory hierarchy compression techniques must contend with a key challenge: supporting *random, fine-grained memory accesses*. Whereas classic compression techniques work best on large blocks of data, many programs access small amounts of data (words or cache lines) at a time, so compressing data in large chunks would be inefficient. The need for random accesses introduces three interrelated problems. First, it limits memory hierarchy compression techniques to use *small blocks*, on the order of a cache line (64–128 B). Second, because the startup latency of decompressing and compressing a block cannot be amortized across large blocks, it limits these techniques to use *simple compression algorithms* optimized for latency rather than throughput. Third, compressed blocks have *variable size*, requiring an extra level of indirection to translate uncompressed addresses into compressed blocks. Depending on the implementation, this extra level of indirection either requires significant metadata (e.g., extra cache or TLB state) or causes internal fragmentation. These problems limit compression ratio and thus the overall value of the techniques.

Prior work has addressed these issues within the context of cache hierarchies, and thus focuses on compressing cache lines. For example, compression algorithms like BDI [35] and BPC [24] achieve low latency by exploiting redundancy within words of a cache line, and compressed main-memory organizations like LCP [34] achieve low access latency at the expense of lower compression ratios by forcing most cache lines in a page to have the same compressed size. These approaches work well on *array-based applications*, such as scientific workloads, where most data follows a regular layout and uses homogeneous data types.

By contrast, existing compressed hierarchy techniques are ineffective on *object-based applications*, i.e., those where most data is stored in objects. These applications do not have a regular memory layout: each object has fields of different types and compressibilities; objects of different types are interleaved in memory; and objects are *not aligned* to the fixed-size cache lines that compression techniques work with. For these reasons, the evaluations of these techniques show small improvements on object-heavy applications.

We present the first object-based compressed memory hierarchy. We leverage two key insights. First, we observe that objects, not cache lines, are the natural unit of compression for object-based programs. Objects are small, typically on the order of a cache line. And object-based applications follow a disciplined access pattern: they always access data within an object and dereference object pointers to access other objects. Therefore, *compressing variable-size objects* instead of fixed-size cache lines and pointing directly to compressed objects can avoid the extra level of indirection and layout issues of existing compressed main memories. Second, we observe that there is significant redundancy (i.e., commonality or value locality [40]) *across objects of the same type*. Exploiting this redundancy, which current algorithms do not leverage, can enable high compression ratios.

We present two contributions that leverage the above insights to compress object-based applications effectively:

- **Zippads** is a novel compressed object-based memory hierarchy. Zippads transfers objects (rather than cache lines) across levels and transparently compresses them when appropriate. Unlike prior designs, Zippads *does not add a level of translation* between compressed and uncompressed addresses. Instead, Zippads directly rewrites pointers to objects as it moves objects across hierarchy levels. To achieve this, Zippads builds on Hotpads [19, 52], a recent object-based memory hierarchy. Though they are not our focus, Zippads also works well on array-based applications.
- **Cross-object-compression (COCO)** is a novel compression algorithm that exploits redundancy across objects cheaply. COCO chooses a small number of *base objects*, and stores only the bytes that differ from an object's base. While Zippads can use other compression algorithms (e.g., BDI), using COCO increases compression ratio substantially.

We evaluate these techniques in simulation and prototype COCO in RTL. Our evaluation shows that, across a mix of array-based and object-based workloads, these techniques substantially outperform a combination of a state-of-the-art compressed last-level cache and compressed main memory [33]: Zippads alone increases compression ratio by 1.37× on average and by up to 1.54×, and Zippads+COCO increases compression ratio by 1.63× on average and by up to 2×. As a result, Zippads+COCO reduces main memory traffic by 56% and improves performance by 17% on average, while incurring only 3.2% storage overhead.

2 Background and motivation

We first review related work in compressed memory hierarchies, then illustrate the challenges and opportunities of compressed hierarchies in object-based programs.

2.1 Related work in compressed hierarchies

Much prior work has focused on compressed memory hierarchies to reduce data movement. While compression is too onerous to be used in small private caches, it is sensible to implement in main memory and the large last-level cache (LLC). Prior techniques thus can be broadly classified into three domains: *(i)* compression algorithms, *(ii)* compressed memory architectures, and *(iii)* compressed cache architectures.

Compression algorithms aim to reduce the number of bits required to represent a data chunk (e.g., a cache line). Since decompression latency adds delay to the critical path of a memory access, unlike general compression algorithms, memory hierarchy compression favors simpler algorithms that achieve low decompression latency and area overhead, even if they achieve a lower compression ratio. Moreover, since programs issue fine-grained memory accesses, prior hardware-based compression techniques focus on compressing cache lines, matching the natural data transfer granularity of the LLC and main memory.

Frequent pattern compression (FPC) [2] recognizes repeated patterns or small-value integers and uses a static encoding to compress every 32-bit data chunk in a cache line. Base-Delta-Immediate (BDI) [35] observes that values in a cache line usually have a small dynamic range, so BDI compresses a cache line by representing it with a base value and per-word deltas. SC^2 [5] uses Huffman coding to compress cache lines, and recomputes the dictionary infrequently, leveraging the observation that frequent values change rarely. Because recomputing the dictionary requires recompressing all the data, SC^2 is suitable for caches but less attractive for main memory. FP-H [4] is tailored to compress floating-point values. HyComp [4] combines multiple compression algorithms in a single system and dynamically selects the appropriate algorithm. Bit-Plane Compression (BPC) [24] targets homogeneous arrays in GPGPUs and improves compression ratio over BDI by compressing the deltas better.

These techniques add few cycles to each memory access. However, they usually exploit redundancy within a single block, a very fine-grained size. They work well for array-based programs with homogeneous data types. But as we will see later, object-based programs have lower redundancy across nearby words, making these techniques less effective.

Compressed main memory architectures are faced with one key challenge: choosing a memory layout that adds little latency while enabling good compression ratios.

MXT [51] compresses 1 KB data chunks with a heavyweight compression algorithm. While it achieves a high compression ratio, its decompression latency is very high (64

cycles). To locate the compressed data, MXT adds a TLB-like translation table to translate chunk addresses, which adds even more latency and requires significant state.

Robust Memory Compression (RMC) [17] and Linearly Compressed Pages (LCP) [34] trade off lower compression ratios for lower latency overheads. They compress smaller (64-byte) cache lines and leverage the virtual memory system, which they modify to translate from uncompressed virtual pages to compressed physical pages. To keep translation mechanisms simple, each physical page is restricted to be power-of-two-sized. This incurs internal fragmentation, which reduces the compression ratio (e.g., a page that compresses to slightly more than 1 KB must use a 2 KB frame).

RMC and LCP differ in the layout of compressed pages. RMC compresses each cache line to one of four possible sizes. Each page table entry is extended to track the sizes of all the lines (64×2=128 bits). To compute the address of a compressed cache line, the system must add up the sizes of all preceding cache lines, a non-trivial computation [34].

By contrast, LCP requires cache lines to compress to the same size. This makes the compressed cache line address trivial to compute in the common case (a simple shift). LCP stores the non-fitting cache lines uncompressed after all the compressed blocks and chooses the page's compression ratio to minimize the final compressed size.

Other prior work builds on these architectures and improves over them. For example, DMC [25] combines LCP and MXT by applying LCP to hot pages and MXT to cold pages. Compresso [15] introduces techniques to reduce metadata accesses, limit overflows, and improve spatial locality.

Compressed cache architectures have more flexibility than compressed main memory, as their associative structure offers more design choices than main memory's directly addressed layout. The key challenge in compressed caches is tracking compressed lines with small tag array overheads and high data array utilization. These architectures typically perform serial tag and data array accesses, and require extra tag entries (usually 2×, about 6% area overhead) to track more compressed cache lines than uncompressed caches.

VSC [1] divides the cache into sets like in a conventional cache, but lets each set store a variable number of variably-sized, compressed cache lines. Each tag has a pointer to identify the line's location within the set. Cache architectures with decoupled tag and data stores, such as the indirect-indexed cache [22] and V-Way cache [37], use a longer forward pointer and can store compressed cache lines anywhere in the data array, reducing fragmentation. Meanwhile, DCC [42], SCC [41], and DISH [30] leverage decoupled sector caches to track multiple compressed lines per sector.

These compressed caches still compress each cache line individually; prior work that exploits redundancy across lines incurs large overheads [29] and is practical only on throughput-oriented processors with high latency tolerance.

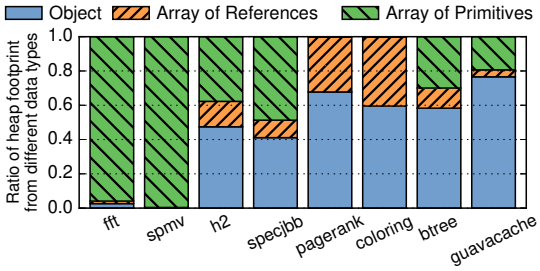


Figure 1. Fraction of the heap consumed by objects and arrays for several Java benchmarks.

2.2 Opportunities for object-based programs

Array-dominated applications, common in scientific computing, provide many opportunities for compression because nearby words share data types and are likely to have similar byte patterns. Prior techniques like BDI [35] and FP-H [4] are effective on these programs. However, many applications are dominated by objects, which have a more irregular layout: nearby words correspond to different fields and are unlikely to have similar values. Prior compression techniques are less effective on object-heavy applications.

Object-heavy applications are common. Fig. 1 shows the footprint composition of eight Java benchmarks. Only the first two benchmarks, which implement common scientific kernels (`fft` and `spmv`), are array-dominated. The remaining six benchmarks, which include databases, graph analytics, and a key-value store, have at least 40% and up to 75% of the heap footprint allocated to objects. More than 90% of those objects are small (< 128 B [8]). Therefore, prior algorithms that leverage similarities among nearby words are unlikely to compress well on these applications because a large portion of their footprint is small objects, not homogeneous arrays.

Fortunately, object-based applications provide new opportunities for compressed memory hierarchies. First, object-based applications perform memory accesses within objects and always follow pointers to other objects. Therefore, objects, and not cache lines, are the right compression unit. Second, though nearby words have low redundancy, similarities exist across objects of the same type.

We now illustrate these two insights with a simple B-tree Java microbenchmark, `BTree`. Fig. 2 shows the three main object types in `BTree`: `Node`, `Entry[]`, and `Entry`, and their in-memory layouts. We show the layout in Maxine [54], the Java Virtual Machine (JVM) we use in our evaluation, but other JVMs like HotSpot [49] and Jikes [3] use a similar layout. Red arrows denote pointers (references) across objects. Fig. 3a shows the memory layout of a 4-entry B-tree node.

Fig. 3b shows an example of applying LCP to `BTree`. LCP uses hybrid BDI+FPC compression (see Sec. 7.1 for details). In this example, each 64B line is compressed into a fixed-size 32B chunk. A few blocks can be compressed beyond 32B, but the remaining space (shown hatched) is left unused due to LCP's design. The total size of this compressed page is then rounded up to the closest power-of-2 page size, which often

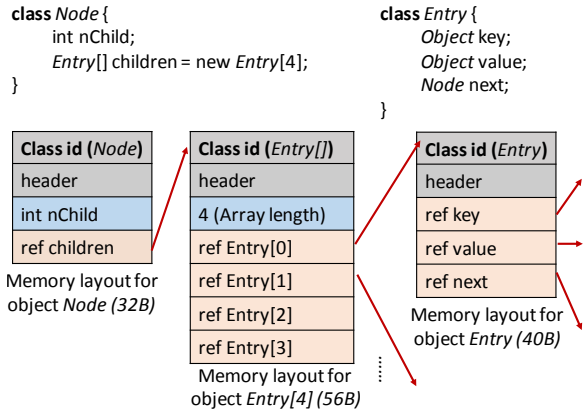


Figure 2. Objects and their memory layout in `BTree`.

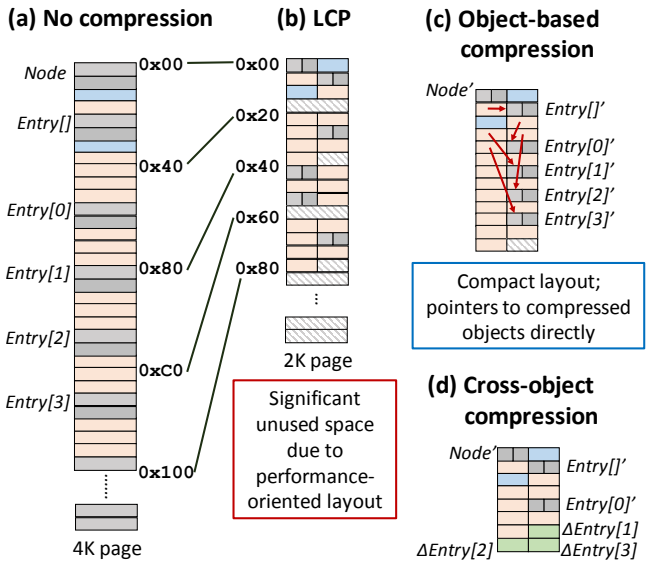


Figure 3. Different compression techniques applied to `BTree`.

causes more unused space, limiting efficiency. For `BTree`, LCP achieves only 10% compression ratio (Table 1), because in addition to layout inefficiencies, the compression algorithm cannot compress many of the objects effectively.

Fig. 3c shows the compressed memory layout if we compress objects instead of cache lines. Compressed objects are stored contiguously, with no space left unused. Moreover, *each pointer (red arrow) directly points to the compressed object.* This approach removes the need to translate between compressed and uncompressed address spaces, and it is safe because programs may access objects only through pointers. This disciplined access pattern removes the fragmentation caused by the tradeoff between compression ratio and fast address calculation in prior work. With this compression technique, we can achieve a compression ratio of 1.56×.

Moreover, there is still an opportunity to improve compression ratio in object-oriented programs. We find that objects of the same type usually have similar contents. For example, many of the bytes in `Entry[0]` and `Entry[1]` are identical. Therefore, it is better to *compress across objects* instead of across nearby words in a cache line.

	No comp. Fig. 3a	LCP Fig. 3b	Object-based Fig. 3c	Cross-object Fig. 3d
Compression ratio	**1.00**	**1.10**	**1.56**	**1.95**

Table 1. Compression ratios of different schemes on `BTree`.

Fig. 3d shows an example of cross-object compression. Instead of storing all `Entry` objects, we store one *base object* (`Entry[0]`). For other `Entry` objects, we only store the bytes that differ (Δ`Entry`) between those objects and the base object. This further reduces footprint over Fig. 3c, achieving an even higher compression ratio of 1.95×.

However, to realize these insights, hardware needs to access data at object granularity and must have control over pointers between objects, as we explain next.

3 Baseline system: Hotpads

Zippads and COCO compress objects instead of cache lines. Thus, they are better suited to an object-based memory hierarchy than a conventional cache hierarchy. We implement them on top of Hotpads, a recent object-based hierarchy. We now describe the principles and relevant features of Hotpads; please see prior work [19, 52] for details.

Modern languages like Java and Go adopt an *object-based* memory model and hide the memory layout from the programmer. This prevents many classes of errors and enables automatic memory management. Hotpads extends the same insight to the memory hierarchy: It hides the memory layout from software and dispenses with the conventional flat address space interface. Instead, Hotpads adopts an object-based interface. Hotpads leverages this interface to efficiently transfer and manage variable-sized objects instead of fixed-size cache lines. Hotpads also provides hardware support for memory allocation, unifies hierarchical garbage collection and data placement, and avoids most associative lookups.

Hotpads is not specific to particular languages, and it is not just a way to accelerate garbage collection or other managed-language features. Rather, Hotpads achieves a more efficient memory system by leveraging the principles behind garbage collection and matching them to the structure of the memory hierarchy. As a result, Hotpads can also accelerate applications in low-level unmanaged languages. These applications can use Hotpads's object-based model selectively, as Hotpads includes a legacy mode to support a flat address space. As we will see in Sec. 7.5, Hotpads and Zippads outperform high-performance allocators on two C/C++ benchmarks.

3.1 Hotpads overview

Hotpads is a hardware-managed hierarchy of scratchpad-like memories called *pads*. Pads are designed to store variable-sized objects efficiently. Hotpads transfers objects across pad levels implicitly, in response to memory accesses. Fig. 5 shows an example Hotpads hierarchy with three levels of pads, but Hotpads can use any number of levels.

Fig. 6 shows the structure of a pad. Most space is devoted to the *data array*, which is managed as a circular buffer. The

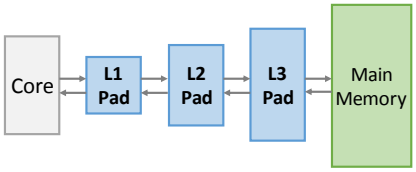


Figure 5. Hotpads is a hierarchical memory system with multiple levels of *pads*.

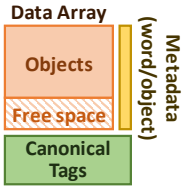


Figure 6. Pad organization.

data array has a contiguous block of *allocated objects* followed by a block of *free space*. Hotpads uses simple *bump pointer* allocation: fetched or newly allocated objects are placed at the end of the allocated region. Pads have two auxiliary structures: *(i)* the *canonical tag (c-tag) array*, which is a decoupled tag store that a fraction of the accesses use to find a resident copy of an object; and *(ii)* the *metadata array*, which tracks information of objects stored in the data array.

Unlike caches, pads have separate addresses from memory and can act as the backing store of some objects. This enables an efficient object flow: objects are first allocated in the L1 pad and *move up* the hierarchy as they become cold and are evicted. Short-lived objects are garbage-collected before they reach main memory, which greatly reduces memory traffic and footprint. An object's *canonical level* is the largest hierarchy level an object it has reached. The canonical level acts as the object's backing store.

3.2 Hotpads example

Fig. 4 illustrates the main features of Hotpads through a simple example showing a single-core system with two levels of pads (we use a single-core setup for simplicity, but Hotpads supports multi-core systems [52]). This example uses only one type of object, `ListNode`, with two members, an integer `value` and a pointer to another `ListNode`.

⓪ shows the initial state of the system: the core's register file holds a pointer to object A in the L2 pad, and A points to object B in main memory. The L1 and L2 pads also hold other objects (shown in solid orange) that are not relevant here.

In this example, A's canonical level (i.e., its backing store) is the L2: A does not exist in main memory and does not have a main memory address. B's canonical level is main memory.

① shows the state of the system after the program accesses A's `value`. A is copied into the L1 pad, taking some free space at the end of the allocated region. Then, the pointer in register `r1` is *rewritten* to point to this L1 copy. This way, *subsequent dereferences of `r1` access the L1 copy directly*.

Programs can also access objects by dereferencing pointers to them. ② shows the state of the system after the core dereferences A's pointer to B. B is copied into the L1 pad, and A's pointer to B is rewritten to point to its L1 copy.

Since programs may have multiple pointers to the same object, pads must have a way to find copies of objects from higher levels. This is the role of the c-tag array, which, for each object copy, stores the object's canonical address (i.e., its address at its canonical level). For example, when A is copied into the L1 pad, the c-tag array inserts a translation from A's L2 address to the copy's L1 address. Thanks to pointer rewriting, only pointers to higher levels need to check the c-tag array. This eliminates most associative lookups.

③ shows the state of the system after the program creates a new object C. C is allocated directly in the L1 pad's free space and requires no backing storage in main memory.

When a pad runs out of free space, it triggers a *collection-eviction* (CE) process to free up space. In ④ the L1 pad has filled up, so the pad starts a CE to free L1 space. Similarly to garbage collection (GC), a CE walks the data array to detect live vs. dead objects. In addition to GC, a CE evicts live but non-recently accessed objects to the next-level pad. In this example, C is dead (i.e., unreferenced) and a new object D is referenced from A, and thus live. Note that A's L1 copy has been modified, so the L2 data is now stale. Only B has been accessed recently in the L1.

⑤ shows the state after the L1 CE. First, C has been collected. Second, A and D have been evicted to the L2 pad. Since A was originally copied from the L2 pad, the modified

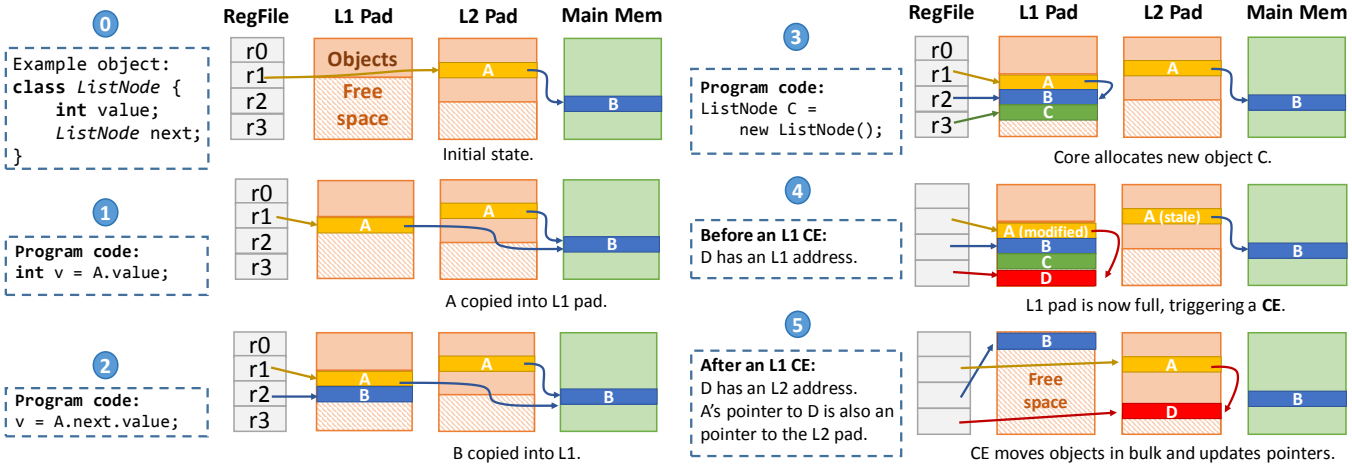


Figure 4. Example showing Hotpads's main features.

Instruction	Format	Operation
Data Load	`ld rd, disp(rb)`	`rd <- Mem[EffAddr]`
Data Store	`st rd, disp(rb)`	`Mem[EffAddr] <- rd`
Pointer Load	`ldptr rp, disp(rb)`	`rp <- Mem[EffAddr]`
Pointer Store	`stptr rp, disp(rb)`	`Mem[EffAddr] <- rp`

Allocation	`alloc rp, rs1, rs2`
	(`rs1` = size)
	(`rs2` = type id)
		`NewAddr <- Alloc(rs1);`
		`Mem[NewAddr] <- rs2;`
		`rp <- NewAddr;`

Table 2. Hotpads ISA. `rd`/`rs` denote dst/src registers that hold data; `rp`/`rb` hold pointers. All accesses use base+offset addressing.

copy is written back to A's L2 location. By contrast, D is moved up to the L2 pad and thus has a new canonical address, an L2 address. Third, B has been kept in the L1 and moved to the start of the array.

As in compacting GC, during CEs, live objects are compacted into a contiguous region to simplify free space management. Moreover, CEs also update pointers in the system (register file, pointers in pads) to point to the new location. For example, both a register (`r3`) and the pointer in A are updated to D's new canonical address.

This always-moving-up object flow is critical for Zippads, as objects start their lifetime uncompressed and move to compressed levels only when they become cold and are evicted. This move changes the object's original address and requires updating all the pointers to the object. Zippads leverages this to point directly to the compressed object, avoiding uncompressed-to-compressed address translation (Sec. 4.1).

3.3 Hotpads implementation details

Next, we discuss the implementation details of Hotpads that are relevant to Zippads and COCO.

ISA: Hotpads modifies the ISA to support an object-based memory model. These ISA changes are transparent to application programmers, but require runtime and compiler modifications. Table 2 shows a subset of the ISA to support object accesses and to convey pointer information to Hotpads. Pointers may be dereferenced or compared, but their raw contents cannot be accessed. This lets Hotpads control their contents. Hotpads uses base+offset addressing modes, where *the base register always holds an object pointer*. The offset can be an immediate (base + displacement mode) or a register (base + index mode). Data load/store instructions are used for non-pointer data (1); pointer load/store instructions are used to access pointer data (2); and the `alloc` instruction allocates a new object (3).

Pointer format: Hotpads controls and manipulates pointers within pads. Fig. 7 shows the format of Hotpads pointers. This format is a microarchitectural choice, as pointers are opaque to software. The lower 48 bits contain the object's address and always point to the first word of the object. The upper 14 bits contain the object's size (in words), and the other 2 bits store metadata that is not relevant to Zippads. All objects are word-aligned. Storing the object's size in the pointer simplifies reading objects: fetching `size` words from the starting address yields the entire object. Zippads extends this pointer format to store compression metadata.

Figure 7. Hotpads pointer format.

Collection-evictions: CEs occur entirely in hardware, and are much faster than software GC because pads are small. To make CEs efficient, Hotpads enforces an important invariant: Objects at a particular level may only point to objects at the same or higher levels. In this way, CEs at smaller pads (e.g., L1) will not involve larger pads (L2, L3) because those pads have no pointers to the L1 pad. This makes CE cost proportional to pad size.

CEs enable Hotpads's object flow: evicting an object from its original canonical level to the next level requires updating all the pointers to the object (e.g., D from 4 to 5). This would be impractically expensive to do for a single object, requiring a scan of the evicting pad and all smaller ones. But CEs amortize this scan across all evicted objects, making updating pointers cheap. Zippads thus piggybacks on CEs to point to compressed objects directly (Fig. 3c).

4 Zippads: An object-based compressed memory hierarchy

Zippads leverages Hotpads to *(i)* manipulate and *compress objects rather than cache lines*, and *(ii)* avoid the extra level of indirection in conventional compressed main memories by *pointing directly to compressed objects*. Zippads is agnostic to the compression algorithm used, and can use conventional algorithms like BDI or FPC, but works best with the COCO compression algorithm. We first describe Zippads, then explain COCO in the next section.

Fig. 8 shows an example Zippads hierarchy. The last-level pad and main memory are compressed, while the core-private L1 and L2 pads are not. Other Zippads hierarchies are possible, e.g., there could be multiple levels of compressed pads, or only main memory could be compressed. Once a level uses compression, it is sensible for larger levels to remain compressed, though Zippads does not require this. Zippads transfers compressed objects directly between compressed levels. To simplify our explanation, we first assume objects are always small (< 128 B). We discuss how Zippads handles larger objects in Sec. 4.3.

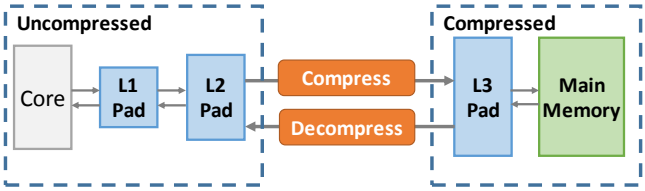


Figure 8. Example Zippads hierarchy with a compressed last-level pad and main memory.

4.1 Compressing objects

Zippads aims to store compressed objects compactly, with no unused space between them to maximize compression ratio. Objects move from uncompressed to compressed storage for two reasons: newly moved objects and dirty writebacks.

Case 1. Newly moved objects: As explained in Sec. 3, Hotpads performs in-hierarchy memory management: objects start their lifetime at the L1 pad, and are moved into larger pads and main memory when they have not been recently used. Hotpads leverages this object flow to minimize the impact of dead objects, collecting them as soon as possible. Zippads further leverages this to facilitate compression: objects start their lifetime uncompressed, and when they become not-recently used, they are evicted into the last-level pad and compressed there. This is a key difference from conventional hierarchies, where objects are mapped to a main memory address to begin with, forcing the problem of translating from uncompressed to compressed addresses.

Newly moved objects are the easiest case to handle: Zippads simply compresses the object and then stores it at the beginning of the available space (with bump-pointer allocation). Fig. 9 illustrates this process. This leaves no space between compressed objects (however, compressed objects are still word-aligned and may have a few unused bytes). The object's new canonical address is now in compressed memory, and all pointers to the object are updated to this new canonical address, as explained in Sec. 3.2.

Case 2. Dirty writebacks: An object can reach to a compressed level, then be fetched into the L1 and modified. This object is then eventually written back to the compressed level. This dirty writeback is more complex than the initial move, because the object's compressed size may change. And since other objects in this level may have pointers to this object, Zippads cannot simply move it.

Fig. 10 shows how Zippads handles dirty writebacks. If the compressed object's new size is the same or smaller than its old size, the object is stored in the old location. If the new size is smaller, this wastes some space, which is left unused.

However, if the compressed object's new size is larger than its old size, the object cannot be stored at its old location. We call this an *overflow*. Zippads allocates a new location for the compressed object (using bump-pointer allocation as usual). Because other pointers to the old location may still exist, Zippads turns the old location into a *forwarding thunk*: it stores the new pointer in the first word of the object. Further accesses to the old location follow the forwarding thunk to find the object. As we will see, overflows are rare (Sec. 7.6).

Periodic compaction: Although dirty writebacks that change the size of an object are rare, they introduce storage inefficiencies, either leaving some unused space or causing forwarding thunks. However, these inefficiencies are temporary: in compressed pads, the collection-eviction (CE) process compacts

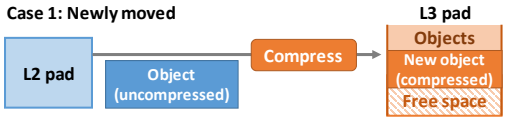


Figure 9. Compressing newly moved objects.

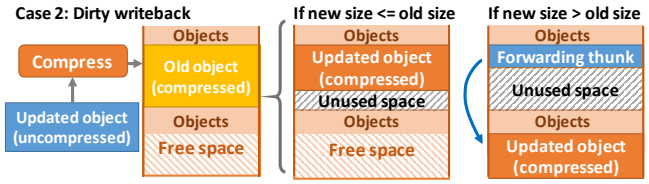


Figure 10. Compressing objects on dirty writebacks.

all the live objects into a contiguous region. Zippads modifies the compaction step of CEs to handle recompressed objects: it eliminates both the unused space at the end of smaller recompressed objects and the forwarding thunks caused by overflows. Like Hotpads, Zippads performs main-memory garbage collection in software. Zippads enhances the garbage collection algorithm to work on compressed objects and to perform these compaction optimizations.

4.2 Encoding compression information in pointers

Most compressed cache architectures employ a common optimization: they use the cache tag to encode information about the compressed cache line needed to perform accesses and decompression, such as the compressed size or the type of compression algorithm it uses. Leveraging the tags is more efficient than encoding this information in the data array itself, as the cache immediately knows how much data to access and what decompression algorithm to use.

Zippads cannot use this optimization, most importantly because main memory has no cache tags, but also because not all pad objects may have a canonical tag. Instead, Zippads *encodes compression information directly in pointers*. This is possible because pointers are opaque to software and we can change their format without changing the ISA. This approach yields all the benefits of encoding information in tags because all accesses start from a pointer.

Fig. 11 shows Zippads's pointer format. First, the size field now encodes the object's *compressed size*. Second, Zippads devotes a few extra bits to store algorithm-specific compression information (e.g., the compression format used). This way, when transferring objects between levels, hardware knows how much data to fetch and which decompression algorithm to use. Moreover, this encoding enables using different compression schemes for different objects.

Algorithm-specific compression information slightly reduces the address width. This is acceptable because Zippads uses word (i.e., 8-byte) addresses. For example, when Zippads

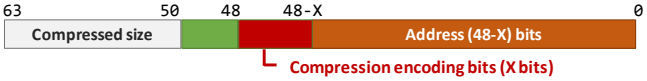


Figure 11. Zippads pointer format. Compression information is encoded in the pointer.

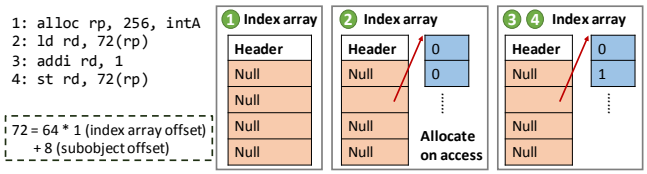


Figure 12. Zippads breaks large objects into subobjects.

uses the BDI algorithm, which requires 4 bits per pointer, 44-bit word addresses allow almost the same address space as conventional 48-bit byte addresses in x86-64.

4.3 Compressing large objects

So far, we have assumed that all objects are small (<128B). However, programs can allocate larger objects. In Hotpads, large objects and arrays are fetched as subobjects in 64B chunks to avoid overfetching (e.g., if only one element is used in a 1KB array). Zippads also needs to handle large objects; otherwise, decompressing a large object for just one element would incur a very high latency.

Zippads thus breaks large objects to smaller subobjects and compresses them individually. This way, when the core only accesses part of a large object, Zippads need not decompress the whole object. Specifically, when allocating a large object (>128B in our implementation), Zippads does not reserve the full capacity for it. Instead, it first allocates an array of pointers, which we call the *index array*. Each pointer in the index array points to a 64B subobject. All pointers are initially null, and the space for a subobject is allocated when an access to a particular subobject occurs (i.e., allocate-on-access).

Fig. 12 shows an example of allocating and accessing a large object. At 1, the program allocates a 256B object, so Zippads allocates an index array with 4 elements. At 2, the core accesses the element at a 72-byte offset, which belongs to the second subobject. Zippads then allocates a subobject and modifies the pointer in the index array. At 3 and 4, the core updates the value and writes it back to the location. Zippads first accesses the index array to find the subobject pointer and traverses this pointer to update the field.

The index array is a microarchitecture optimization invisible to software. Subobjects are also compressed when their canonical addresses change. Pointers in the index array are updated as normal objects in Hotpads. Compressing at subobject granularity also avoids moving large objects for overflows. One drawback is that the footprint of large objects increases by $\frac{1}{8}$, but this is a small overhead compared to the benefit of low decompression latency and a more compact layout. Evaluation results show that Zippads offsets this overhead (Sec. 7.2).

5 COCO: Cross-object-compression algorithm

Zippads works with any compression algorithm. But there is limited redundancy within each object, so existing algorithms like BDI yield limited benefits. We thus propose

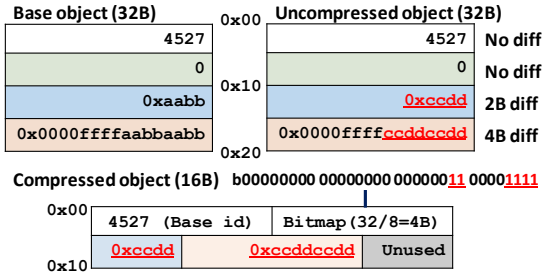


Figure 13. Example COCO-compressed object.

COCO (Cross-Object COmpression), a new compression algorithm that exploits redundancy across objects. COCO achieves high compression ratios and is cheap to implement.

5.1 COCO compression format

COCO is a differential compression algorithm: it compresses an object by storing only the bytes that differ from a different *base object*. Specifically, the compressed object format has three elements:

1. The *base object id*, an integer (32 bits in our implementation) that uniquely identifies the base object.
2. A *diff bitmap* with one bit per byte of the object. The i^{th} bit is set iff the object's i^{th} byte differs from the base object.
3. A string of *byte diffs* containing the bytes that are different from the base object's.

Fig. 13 shows an example COCO-compressed object. The uncompressed object has the same values in the first and second words, a 2-byte difference in the third, and a 4-byte difference in the fourth. Therefore, the compressed object stores only the six differing bytes in addition to the header.

5.2 Compression and decompression circuits

COCO compression/decompression circuits are simple to implement and only require narrow comparators and multiplexers. Our implementations compress/decompress one word (8 bytes) per cycle. This provides sufficient throughput for our purposes. The compression circuit compares the base object and the uncompressed object word by word. Each cycle, it produces one byte of the *diff bitmap* and a chunk of delta bytes. The decompression circuit takes the base object, bitmap, and delta bytes as inputs and produces one word of the decompressed object per cycle.

We have written the RTL for these circuits and synthesized them at 45nm using yosys [55] and the FreePDK45 standard cell library [23]. The compression circuit requires an area equivalent to 810 NAND2 gates at a 2.8 GHz frequency. The decompression circuit requires an area equivalent to 592 NAND2 gates at a 3.4 GHz frequency. These frequencies are much higher than typical uncore frequencies (1–2 GHz), and a more recent fabrication process would yield faster circuits.

Finally, COCO area overheads are much smaller than prior techniques, such as BPC (68K NAND2 gates [24]) or C-pack (40K NAND2 gates [13]). This result shows that COCO is practical and simple to implement in hardware.

5.3 Building the base object collection

COCO allocates extra space in main memory to store base objects. We empirically find that statically assigning one base object per type id works well: same-type objects have the same layout and often share many values. This approach also makes compression faster: instead of trying multiple base objects to decide which base object to use, COCO simply selects the base object using the object type id (the first word of the uncompressed object).

We find that COCO is largely insensitive to the choice of base object, so our implementation simply uses the first object of each type that it sees (i.e., the first object that is evicted to a compressed level) as the base object. It may be beneficial to dynamically update the base object or to have multiple base objects per type; we leave this to future work.

5.4 Caching base objects

Compressing and decompressing objects require fast access to the base object. If COCO had to access the base object from the last-level pad or main memory on each decompression, this would significantly increase decompression latency and bandwidth consumption.

Instead, we serve base objects from a small and fast *base object cache*, 8 KB in our implementation. This cache stores the most frequently used base objects, and is indexed with the base object id.

We find that a small cache suffices because the popularity of object types is highly skewed. Fig. 14 shows the distribution of accesses over the most popular object types for four selected apps. Tens of object type ids account for most of the accesses.

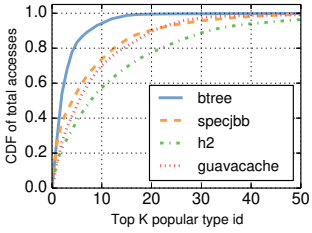


Figure 14. CDF of accesses to most popular object ids.

This overhead is similar to prior dictionary-based compression algorithms, such as SC^2 or FP-H (4 KB dictionary [4, 5]). COCO only needs to fetch the base object content when it misses in the base object cache.

6 Integrating Zippads and COCO

We have so far discussed Zippads independently from the compression algorithm, and COCO independently from Zippads. As discussed earlier, objects and arrays have different types of redundancy, and Zippads should compress both objects and arrays well. Therefore, Zippads uses different compression algorithms for each: COCO for objects, and a conventional hybrid algorithm, BDI+FPC, for arrays. To this end, our Zippads+COCO implementation adds 4 bits of compression metadata to each pointer, as shown in Table 3. The first bit denotes whether the data is an array compressed with BDI+FPC, and the remaining 3 bits are used by BDI+FPC compression. If it is not a compressed array, the second bit indicates whether it is an object compressed with COCO. If it is also not a COCO-compressed object, the third bit indicates whether it is an object or an array, which is required for Zippads+COCO to compress data during CEs.

	Uncomp. Object	Uncomp. Array	Comp. Object	Comp. Array
Code	**000X**	**001X**	**01XX**	**1CCC**

Table 3. Zippads+COCO in-pointer compression information. X denotes the bit does not matter, and Cs denote the bits used by hybrid BDI+FPC encoding.

Instruction	**Format**	**Operation**
Allocate Array	`alloc_array rp, rs1, rs2` (`rs1` = size) (`rs2` = type id)	Same as `alloc`, but sets the array bit in `rp`.

Table 4. `alloc_array` lets Zippads distinguish arrays and objects.

We also extend the ISA to distinguish objects and arrays at allocation time: Zippads adds a new `alloc_array` instruction, as shown in Table 4, used to allocate arrays, and leaves the original `alloc` to allocate objects. Both instructions are identical, except that `alloc_array` sets the array bit in the new pointer, whereas `alloc` does not.

Arrays use hybrid BDI+FPC compression, in the same style as HyComp [4]. We use 3 bits in the pointer to select the right decompression algorithm, and replace one choice in the original BDI encoding (Base2-Δ1) to represent FPC compression.

Non-COCO Zippads variant: To better understand the benefits of Zippads and COCO, we also evaluate Zippads-BF, a variant of Zippads that does not use COCO. Zippads-BF instead uses hybrid BDI+FPC for all objects and arrays. As we will see, Zippads-BF outperforms existing compressed hierarchies due to its more compact layout.

6.1 Discussion

Although we have integrated COCO with Zippads in this work, in principle COCO should be usable with other compressed architectures. However, these architectures should somehow convey object boundaries to COCO, which would require runtime and hardware changes. For example, one solution could be to use type-segregated object pools, where each region of memory is dedicated to storing objects of a particular type, instead of bump-pointer allocation; however, this makes object allocation slower, may hurt spatial locality, and requires significant metadata to map pools to object types. Another approach could be to align all objects to cache line boundaries. This could achieve good performance on the compressed cache and memory, but excessive padding would use uncompressed caches (L1 and L2) poorly.

In the end, because retrofitting an object-based compression algorithm into a cache-based hierarchy faces significant hurdles and Zippads already demonstrates significant improvements over prior techniques even without COCO, we choose to not implement COCO outside of Zippads.

Core		x86-64 ISA, 3.6 GHz, Westmere-like OOO [39]: 16B-wide ifetch; 2-level bpred with 2k×10-bit BHSRs + 4k×2-bit PHT, 4-wide issue, 36-entry IQ, 128-entry ROB, 32-entry LQ/SQ
Caches	**L1**	64 KB, 8-way set-associative, split D/I caches, 64 B lines
	L2	512 KB private per-core, 8-way set-associative
	LLC	4 banks, 2 MB/bank, 16-way set-associative, LRU
	Mem	2 DDR3-1600 channels
CMH	**Algo**	HyComp-style hybrid [4]: BDI [35] (1-cycle latency) and FPC [2] (5-cycle latency)
	LLC	2× tag array, VSC [1] cache, CAMP [32] replacement policy
	Mem	LCP [34] with perfect, 32KB metadata cache
Hotpads	**L1D**	64 KB data array, 1K ctag entries
	L1I	64 KB cache, 8-way set-associative, 64 B lines
	L2	512 KB data array, 8K ctag entries
	LLP	4×2 MB data array, 4×32K ctag entries
Zippads		**L3:** 4×64K ctag array, 8 KB base object cache, COCO compression (1-cycle latency) and hybrid BDI+FPC

Table 5. Configuration of the simulated system.

7 Evaluation

We evaluate Zippads and COCO on a mix of array-based and object-based workloads. We evaluate on Java workloads because Java is a memory-safe language that aligns well with our baseline system, Hotpads. To show that Zippads is not specific to Java or memory-safe languages, we also evaluate on two C/C++ benchmarks in Sec. 7.5.

7.1 Methodology

We evaluate Zippads using MaxSim [38], a simulation platform that combines ZSim [39], a Pin-based [27] simulator, and Maxine [54], a 64-bit metacircular research JVM.

7.1.1 Hardware

We compare the following techniques:

Uncompressed: Our baseline uses a three-level cache hierarchy without compression, with parameters shown in Table 5.

Compressed memory hierarchy (CMH): We implement a state-of-the-art compressed memory hierarchy that compresses both the LLC and main memory. We use HyComp-style hybrid BDI+FPC compression (Sec. 6). The compressed LLC uses the VSC [1] design with 2× tag array entries and uses the CAMP compression-aware replacement policy [32]. The compressed main memory uses LCP [34], which we idealize by assuming a perfect metadata cache that always hits.

Hotpads: We configure Hotpads as in prior work [19, 52], with three levels of uncompressed pads.

Zippads: Zippads uses a compressed last-level pad with 2× the canonical tag array entries, similar to the VSC LLC in the compressed memory hierarchy design. We also use an 8KB base object cache to store frequently-used base objects. Zippads uses COCO for objects and BDI+FPC for arrays; we also evaluate a variant of Zippads, Zippads-BF, that always uses BDI+FPC for objects and arrays, like the CMH design.

Java Benchmark	**Input**
fft	2^{21} points
spmv	m = 1M, nonzero = 64M
h2	default input: 4K transactions
specjbb	1 warehouse per thread, 50K transactions
pagerank	amazon-2008 graph
coloring	amazon-2008 graph
btree	ycsb-c, 2M key-value pairs, 4M queries
guavacache	ycsb-c, 4M key-value pairs, 4M queries
C Benchmark	**Input**
gcbench	16M nodes, each node is 32B large
silo	tpcc, 1 warehouse, 8K transactions

Table 6. Workloads and inputs used.

Cache scrubbing: Modern languages like Java incur memory overheads due to garbage collection [56]. Therefore, we also implement cooperative cache scrubbing [44], which adds instructions to zero and scrub (i.e., undirty) cache lines and use them in the JVM for both the uncompressed and compressed cache hierarchies to reduce memory traffic due to object allocation and recycling. Scrubbing does not improve compression ratio, but it improves the performance of garbage-collected languages with simple mechanisms.

7.1.2 Software

JVM: Our cache-based baseline uses the Maxine JVM with a tuned, stop-the-world generational GC with a 64 MB young heap. Hotpads and Zippads use a modified JIT compiler that follows their new ISA.

Workloads: We study eight Java workloads from different domains. We select workloads with heap sizes larger than 100 MB, so that they exercise main memory. We use two scientific workloads, `fft` and `spmv` from the Scimark2 [36] suite; two database workloads, `h2` from the Dacapo [8] suite[1] and `SPECjbb2005` [48]; two graph processing workloads, `PageRank` and `Coloring` from JgraphT [28], a popular Java graph library; `GuavaCache`, a key-value store from Google Core Libraries for Java [21], and `BTree`, the example we saw in Sec. 2, from the JDBM [26] database.

Table 6 describes their input sets. We fast-forward JVM initialization and warm up the JIT compiler like prior work [8] by running the same workload multiple times before starting simulation. We run all workloads to completion to avoid sampling bias in compression ratio [16]. For each workload, we first find the smallest heap size that does not crash, and use 2× that size. This is standard methodology [9, 44].

In addition to Java workloads, we also study two C/C++ workloads. `GCBench` [10] is a C benchmark that creates, traverses, and destroys binary trees. `GCBench`'s default input incurs only a 16MB active memory footprint, so we scale the input to incur a 512MB active memory footprint, which stresses main memory. `Silo` [53] is a C++ in-memory OLTP

[1] We evaluate Java workloads with large footprints (>100MB min heap size); h2 is the only such one from DaCapo.

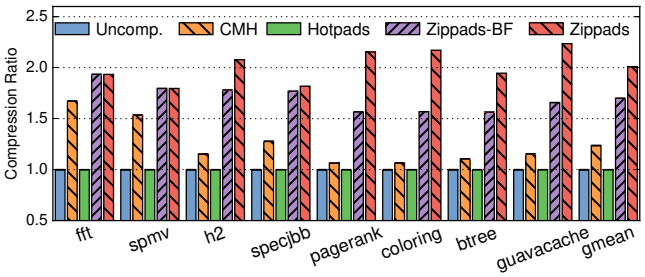


Figure 15. Compression ratio of different schemes.

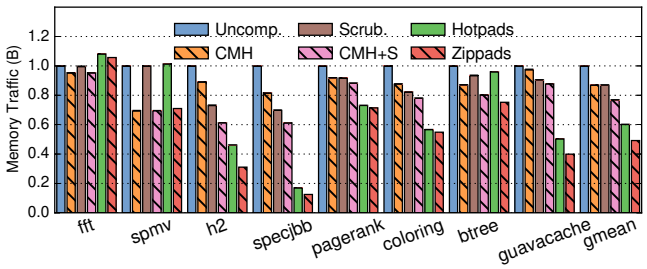


Figure 16. Normalized main memory traffic of different schemes.

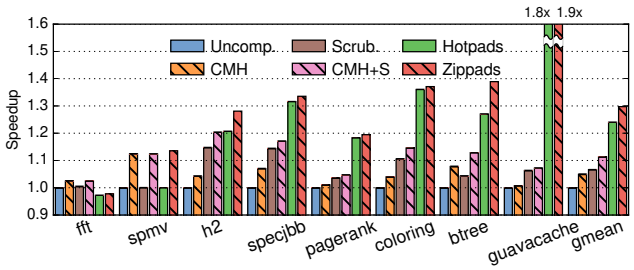


Figure 17. Performance of different schemes.

database, configured to run the TPC-C benchmark. Both the baseline and CMH use high-performance allocators (tcmalloc [20] for `GCBench` and jemalloc [18] for `Silo`). For Hotpads and Zippads, we modify these workloads to use the Hotpads and Zippads ISAs to allocate and access objects.

Metrics: We report the average compression ratio, sampled every 100M cycles, of different schemes. We also report total memory traffic (in bytes) and performance (inverse of execution time). All metrics are normalized to the baseline system without compression.

7.2 Zippads improves compression ratio

Fig. 15 shows the compression ratio of the five memory hierarchies we compare. Each group of bars shows results for a different application. Compressed hierarchies have their bars hatched; uncompressed hierarchies are shown unhatched.

CMH compresses memory footprint effectively for array-dominated scientific workloads `fft` and `spmv`, achieving compression ratios of 1.67 and 1.53, respectively. Zippads also compresses these two workloads well and achieves slightly higher compression ratios than CMH, 1.97 and 1.79, because it better compresses non-heap data (e.g., code and JVM state), which is not array-based. There is no difference between Zippads-BF and Zippads because these two workloads are dominated by arrays, which Zippads always compresses with hybrid BDI+FPC.

The other workloads are object-dominated, and differences across techniques are larger. CMH only compresses around 10% of the total footprint and has compression ratios between 1.06 to 1.27. By contrast, Zippads achieves high compression ratios. The difference in compression ratio between CMH and Zippads correlates well with the ratio of object footprint shown in Fig. 1. For example, `guavacache` has the highest ratio for objects in the main memory footprint, and the difference between CMH and Zippads is also the highest: Zippads compresses 2× better than CMH. Meanwhile, `specjbb` has the lowest ratio of objects (around 40%) in the heap footprint, so the difference in compression ratio between CMH and Zippads is also the lowest among these workloads.

There is also a large difference between Zippads-BF and Zippads in these applications. Owing to its more compact layout, Zippads-BF achieves compression ratios of 1.56–1.78 in these applications, significantly higher than CMH, despite only using BDI+FPC. By contrast, Zippads uses COCO for objects, which increases compression ratios to 1.82–2.24.

On average, CMH achieves a compression ratio of only 1.24, while Zippads-BF and Zippads achieve ratios of 1.70 and 2.01, i.e., 1.37× and 1.63× better than CMH. These results show that compressing objects rather than cache lines and adopting an object-specific compression algorithm are both important contributions to the effectiveness of Zippads.

7.3 Zippads reduces main memory traffic

Fig. 16 shows the memory traffic of all schemes, measured in total bytes read and written, normalized to Uncompressed. In addition to schemes we saw above, Fig. 16 also reports the Scrubbing variants of Uncompressed and CMH (CMH+S).

We find that CMH reduces main memory traffic for many applications for two main reasons. First, the compressed LLC has a higher effective capacity, and thus higher cache hit rate. This helps cache-capacity sensitive applications, in particular `h2`, `specjbb`, and `btree`. Second, the compressed main memory (LCP) lets the system fetch consecutive cache lines in a single 64-byte burst, which helps applications with high spatial locality. This is the case for `spmv`, `coloring`, and `pagerank`. On average, CMH reduces main memory traffic by 15% over Uncompressed.

Scrubbing helps the allocation-heavy database workloads, reducing their traffic by 60%. But Scrubbing is not as effective for others, especially for scientific workloads that only allocate once. On average, Scrubbing reduces main memory traffic by 15%. CMH and Scrubbing (CMH+S) yield additive benefits since they are orthogonal techniques. CMH+S saves 30% of main memory traffic on average.

Hotpads does not reduce traffic for scientific workloads, but it saves memory traffic significantly for object-based workloads because of its object-friendly features: object-gra-

nularity data movement, in-pad allocation, and hardware-based in-pad garbage collection. These features especially help h2, specjbb, and guavacache. On average, Hotpads reduces main memory traffic by 66%.

Zippads improves over Hotpads by adding the benefits of high compression ratios. In workloads where CMH saves significant traffic, such as spmv, h2, and btree, Zippads also yields significant benefits over Hotpads. Like Hotpads, Zippads also benefits guavacache significantly, whereas other techniques yield little benefit. On average, Zippads reduces main memory traffic by 2× over the baseline, by 56% over CMH+S, and by 22% over Hotpads.

7.4 Zippads improves system performance

Fig. 17 shows the end-to-end performance of the different memory hierarchies. The performance improvement of different schemes correlates well with their memory traffic reduction. For example, CMH reduces the memory traffic for spmv, and it also improves performance by 12%. Scrubbing reduces memory traffic the most for database workloads and thus improves performance the most for them.

Hotpads's object-level operation and in-pad memory management provide large benefits across all object-based programs. Zippads again adds the benefits of memory compression over Hotpads, helping spmv, h2, and btree.

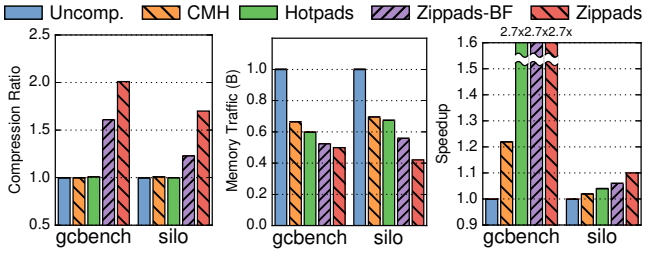
On average, CMH improves performance by 5%, Scrubbing by 6%, CMH+S by 11%, and Hotpads by 24%. Zippads improves performance over Uncompressed by 30%. This represents a 5% improvement over Hotpads and a 17% improvement over CMH+S. Overall, these results show that Zippads incurs small compression overheads, so compression consistently improves performance.

7.5 Zippads is effective on C/C++ benchmarks

Zippads is not specific to Java workloads, and also helps object-based programs in unmanaged languages. To show this, Fig. 18 compares the compression ratio, normalized main memory traffic, and performance of the different schemes on two object-heavy C/C++ workloads. Small objects occupy over 95% of the memory footprint in these workloads, so trends are similar to those of object-heavy Java workloads.

First, CMH achieves negligible memory footprint reductions for these workloads. By contrast, Zippads-BF achieves high compression ratios for these workloads, 1.61 and 1.23, thanks to its compact layout; and Zippads achieves even higher compression ratios, 2.01 and 1.70, thanks to COCO.

Second, all compression techniques reduce main memory traffic. CMH's reduced memory traffic stem from accesses to freshly allocated pages. These pages are zeroed and thus compress well. Thus, CMH reduces main memory traffic by 47% even though it does not compress the data produced by these workloads. Hotpads reduces main memory traffic by 57%, and Zippads-BF and Zippads reduce traffic further, by 85% and 2.2×, by effectively compressing main memory.



(a) Compression ratio. (b) Memory traffic. (c) Performance.

Figure 18. Results for C/C++ benchmarks.

Finally, CMH achieves a 20% speedup over the uncompressed baseline in GCbench due to its reduced memory traffic. Hotpads and Zippads are both 2.7× faster than the baseline. These speedups stem from Hotpads features, including in-hierarchy allocation and collection-evictions. Zippads preserves Hotpads's high performance while achieving a high compression ratio. Silo is neither memory-intensive nor allocation-intensive, so memory hierarchy compression has modest performance benefits: CMH only improves performance by 4%, Zippads-BF by 6%, and Zippads by 10%.

7.6 Zippads analysis

Base object cache misses: Fig. 19 shows the number of misses per kilo-cycle (MPKC) in the COCO base object cache. Most workloads have fewer than 0.001 MPKC. The database workloads have slightly more misses, but are still around 0.01 MPKC. These misses reduce performance by less than 0.1%. Moreover, a 16KB base object cache does not help much. We thus conclude that a small base object cache is effective.

Overflow frequency: Fig. 20 shows the rate of overflows due to dirty writebacks, in overflows per thousand cycles. Overflows are rare across all workloads we evaluate. They happen most frequently in guavacache, but still at a low frequency of 0.4 overflows per Kcycle. For other workloads, overflows happen less than 0.01 times per Kcycle.

Hardware overhead analysis: Table 7 shows the total storage per last-level cache or pad of different schemes. Hotpads adds 6.5% storage over the baseline for the pad metadata and canonical tag entries. CMH adds 12.7% over the baseline due to 2× tags, encoding bits in tag entries, and a 32KB metadata

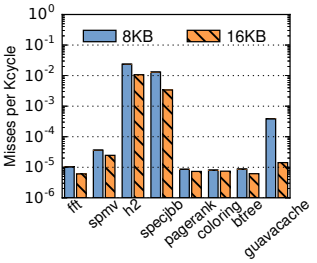


Figure 19. Rate of base object cache misses (in misses per Kcycle, log scale).

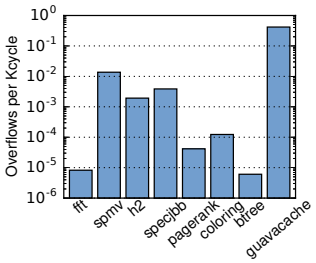


Figure 20. Rate of dirty writeback overflows (in overflows per Kcycle, log scale).

	Tag	Data	Extra	Total (KB)	Increased by (%)
Baseline	160	2048	0	2208	0%
Hotpads	208	2048	96	2352	**6.5%**
CMH	408	2048	32	2488	**12.7%**
Zippads	416	2048	96 + 8	2568	**16.3%**
			9.2% over Hotpads		**3.2% over CMH**

Table 7. On-chip storage (KB) per last-level cache/pad bank.

cache. Similarly, Zippads adds 9.2% over Hotpads due to doubling the number of canonical tags (to track more objects) and the 8KB base object cache. This is only 3.2% extra storage over the CMH LLC and 16.3% over the baseline. Overall, this shows similar on-chip storage requirements as prior compressed caches. These overheads are uniformly offset by the high compression ratios that Zippads achieves. Moreover, most overheads stem from the larger tag array, which can be removed if only compressed main memory (2x smaller memory footprint) is needed.

8 Additional related work

Prior work on software techniques to reduce the memory footprint of managed languages has also considered object-level compression. They compress objects by removing zeros [12, 45] or frequent field values [11]. COCO is inspired by these techniques. However, prior software techniques must be simple and must be used selectively to limit overheads. By contrast, COCO can be used to compress all objects.

Discontiguous array designs [12, 43] divide arrays into indexed chunks to avoid fragmentation. Zippads's subobject compression shares the same motivation.

Prior work in hardware deduplication [14, 47, 50] also seeks to reduce memory footprint. However, deduplication techniques work well only when applications have coarse-grained redundancy, with large chunks of identical data. COCO can be seen as a byte-level deduplication technique.

Some prior work in compression has also considered bandwidth usage and link utilization. MemZip [46] places compressed cache lines and their metadata for address translation next to each other to reduce bandwidth usage. Toggle-aware compression [31] considers the extra dynamic energy consumed by on-chip and off-chip links due to the more frequent toggling caused by compression. Zippads can be combined with these techniques (e.g., data bus inversion) to further reduce the energy consumed by links.

9 Conclusion

Conventional compressed hierarchies focus on compressing cache lines, which limits compression efficiency and adds substantial overheads. In this paper, we leverage two insights about object-based programs to improve compressed hierarchies. First, these programs perform object-level accesses, so objects are the right unit of compression. Second, there is significant redundancy across objects. Using these insights, we propose Zippads, the first object-based compressed memory hierarchy, and COCO, a new, cross-object-compression algorithm. Zippads+COCO improves compression ratio over a combination of state-of-the-art techniques by up to 2× and by 1.63× on average. It also reduces memory traffic by 56% and improves performance by 17%.

Acknowledgments

We sincerely thank Maleen Abeydeera, Joel Emer, Mark Jeffrey, Anurag Mukkara, Suvinay Subramanian, Victor Ying, and the anonymous reviewers for their feedback. We thank Nathan Beckmann for sharing his compressed cache and CAMP implementation [6, 7], and Gennady Pekhimenko for sharing his BDI and FPC implementations. This work was supported in part by NSF grant CAREER-1452994 and by a grant from the Qatar Computing Research Institute.

References

[1] Alaa R Alameldeen and David A Wood. 2004. Adaptive cache compression for high-performance processors. In *Proc. ISCA-31*.

[2] Alaa R Alameldeen and David A Wood. 2004. *Frequent pattern compression: A significance-based compression scheme for L2 caches.* Technical Report 1500. Dept. Comp. Sci., Univ. Wisconsin-Madison.

[3] Bowen Alpern, Steve Augart, Stephen M Blackburn, Maria Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen Fink, David Grove, Michael Hind, et al. 2005. The Jikes research virtual machine project: Building an open-source research community. *IBM Systems Journal* 44, 2 (2005).

[4] Angelos Arelakis, Fredrik Dahlgren, and Per Stenstrom. 2015. HyComp: A hybrid cache compression method for selection of data-type-specific compression methods. In *Proc. MICRO-48*.

[5] Angelos Arelakis and Per Stenstrom. 2014. SC2: A statistical compression cache scheme. In *Proc. ISCA-41*.

[6] Nathan Beckmann and Daniel Sanchez. 2015. *Bridging Theory and Practice in Cache Replacement.* Technical Report MIT-CSAIL-TR-2015-034. Massachusetts Institute of Technology.

[7] Nathan Beckmann and Daniel Sanchez. 2017. Maximizing Cache Performance Under Uncertainty. In *Proc. HPCA-23*.

[8] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J Eliot B Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. OOPSLA*.

[9] Stephen M Blackburn and Kathryn S McKinley. 2008. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proc. PLDI*.

[10] Hans-J Boehm. 2002. An artificial garbage collection benchmark. http://hboehm.info/gc/gc_bench.html, archived at https://perma.cc/Y4BY-7RN4.

[11] Guangyu Chen, Mahmut Kandemir, and Mary J Irwin. 2005. Exploiting frequent field values in Java objects for reducing heap memory requirements. In *VEE*.

[12] Guangyu Chen, M Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, Bernd Mathiske, and Mario Wolczko. 2003. Heap compression for memory-constrained Java environments. In *Proc. OOPSLA*.

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

18, 8 (2010).

[14] David Cheriton, Amin Firoozshahian, Alex Solomatnikov, John P Stevenson, and Omid Azizi. 2012. HICAMP: Architectural support for efficient concurrency-safe shared structured data access. In *Proc. ASPLOS-XVII*.

[15] Esha Choukse, Mattan Erez, and Alaa Alameldeen. 2018. Compresso: Pragmatic main memory compression. In *Proc. MICRO-51*.

[16] Esha Choukse, Mattan Erez, and Alaa Alameldeen. 2018. CompressPoints: An evaluation methodology for compressed memory systems. *Computer Architecture Letters* 17, 2 (2018).

[17] Magnus Ekman and Per Stenstrom. 2005. A robust main-memory compression scheme. In *Proc. ISCA-32*.

[18] Jason Evans. 2005. jemalloc http://jemalloc.net/.

[19] Yee Ling Gan. 2018. *Redesigning the memory hierarchy for memory-safe programming languages*. Master's thesis. Massachusetts Institute of Technology.

[20] Sanjay Ghemawat and Paul Menage. 2005. TCMalloc: Thread-Caching Malloc http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[21] Google. 2004. Guava: Google Core Libraries for Java. https://github.com/google/guava.

[22] Erik G Hallnor and Steven K Reinhardt. 2005. A unified compressed memory hierarchy. In *Proc. HPCA-11*.

[23] Nangate Inc. 2008. The NanGate 45nm Open Cell Library. http://www.nangate.com/?page_id=2325.

[24] Jungrae Kim, Michael Sullivan, Esha Choukse, and Mattan Erez. 2016. Bit-plane compression: Transforming data for better compression in many-core architectures. In *Proc. ISCA-43*.

[25] Seikwon Kim, Seonyoung Lee, Taehoon Kim, and Jaehyuk Huh. 2017. Transparent dual memory compression architecture. In *Proc. PACT-26*.

[26] Jan Kotek. 2012. JDBM: A simple transactional persistent engine for Java. http://jdbm.sourceforge.net/.

[27] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI*.

[28] Barak Naveh. 2003. JGraphT. http://jgrapht.org.

[29] Tri M Nguyen and David Wentzlaff. 2015. MORC: A manycore-oriented compressed cache. In *Proc. MICRO-48*.

[30] Biswabandan Panda and André Seznec. 2016. Dictionary sharing: An efficient cache compression scheme for compressed caches. In *Proc. MICRO-49*.

[31] Gennady Pekhimenko, Evgeny Bolotin, Nandita Vijaykumar, Onur Mutlu, Todd C Mowry, and Stephen W Keckler. 2016. A case for toggle-aware compression for GPU systems. In *Proc. HPCA-22*.

[32] Gennady Pekhimenko, Tyler Huberty, Rui Cai, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. 2015. Exploiting compressed block size as an indicator of future reuse. In *Proc. HPCA-21*.

[33] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. 2012. *Linearly compressed pages: A low-complexity, low-latency main memory compression framework*. Technical Report SAFARI 2012-002. Carnegie Mellon University.

[34] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. 2013. Linearly compressed pages: a low-complexity, low-latency main memory compression framework. In *Proc. MICRO-46*.

[35] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. 2012. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proc. PACT-21*.

[36] Roldan Pozo and Bruce Miller. 2004. SciMark 2.0. https://math.nist.gov/scimark2/.

[37] Moinuddin K. Qureshi, David Thompson, and Yale N. Patt. 2005. The V-Way cache: Demand based associativity via global replacement. In *Proc. ISCA-32*.

[38] Andrey Rodchenko, Christos Kotselidis, Andy Nisbet, Antoniu Pop, and Mikel Luján. 2017. MaxSim: A simulation platform for managed applications. In *Proc. ISPASS*.

[39] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proc. ISCA-40*.

[40] Somayeh Sardashti, Angelos Arelakis, Per Stenström, and David A Wood. 2015. A primer on compression in the memory hierarchy. *Synthesis Lectures on Computer Architecture* 10, 5 (2015).

[41] Somayeh Sardashti, André Seznec, and David A Wood. 2014. Skewed compressed caches. In *Proc. MICRO-47*.

[42] Somayeh Sardashti and David A Wood. 2013. Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching. In *Proc. MICRO-46*.

[43] Jennifer B Sartor, Stephen M Blackburn, Daniel Frampton, Martin Hirzel, and Kathryn S McKinley. 2010. Z-rays: divide arrays and conquer speed and flexibility. In *Proc. PLDI*.

[44] Jennifer B Sartor, Wim Heirman, Stephen M Blackburn, Lieven Eeckhout, and Kathryn S McKinley. 2014. Cooperative cache scrubbing. In *Proc. PACT-23*.

[45] Jennifer B Sartor, Martin Hirzel, and Kathryn S McKinley. 2008. No bit left behind: The limits of heap data compression. In *Proc. ISMM*.

[46] Ali Shafiee, Meysam Taassori, Rajeev Balasubramonian, and Al Davis. 2014. MemZip: Exploring unconventional benefits from memory compression. In *Proc. HPCA-20*.

[47] Dimitrios Skarlatos, Nam Sung Kim, and Josep Torrellas. 2017. PageForge: A near-memory content-aware page-merging architecture. In *Proc. MICRO-50*.

[48] Standard Performance Evaluation Corporation. 2006. SPECjbb2005 (Java Server Benchmark). https://www.spec.org/jbb2005/.

[49] Sun Microsystems. 2006. Memory management in the Java HotSpot virtual machine. http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf.

[50] Yingying Tian, Samira M Khan, Daniel A Jiménez, and Gabriel H Loh. 2016. Last-level cache deduplication. In *Proc. ICS'16*.

[51] R Brett Tremaine, Peter A Franaszek, John T Robinson, Charles O Schulz, T Basil Smith, Michael E Wazlowski, and P Maurice Bland. 2001. IBM memory expansion technology (MXT). *IBM Journal of Research and Development* (2001).

[52] Po-An Tsai, Yee Ling Gan, and Daniel Sanchez. 2018. Rethinking the memory hierarchy for modern languages. In *Proc. MICRO-51*.

[53] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proc. SOSP-24*.

[54] Christian Wimmer, Michael Haupt, Michael L Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An approachable virtual machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013).

[55] Clifford Wolf. 2012. Yosys Open Synthesis Suite. http://www.clifford.at/yosys/.

[56] Yi Zhao, Jin Shi, Kai Zheng, Haichuan Wang, Haibo Lin, and Ling Shao. 2009. Allocation wall: A limiting factor of Java applications on emerging multi-core platforms. In *Proc. OOPSLA*.