Rethinking the Memory Hierarchy for Modern Languages

Po-An Tsai, Yee Ling Gan, Daniel Sanchez

Originally presented at MICRO 2018

Today's presenter: Fabian Wüthrich

09 April 2020

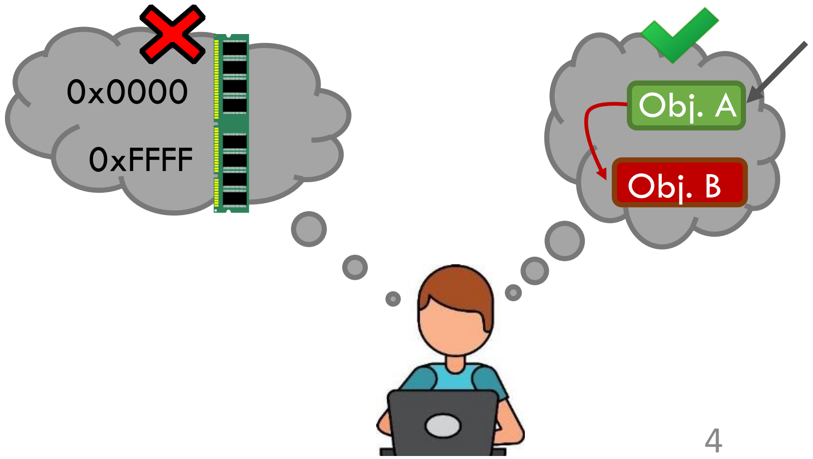
Executive Summary

- **Problem:** flat address space is inefficient for memory-safe languages
  - Move cache lines instead of objects (ignore semantics)
  - Short-lived objects require backing storage in main memory
- **Solution:** Hotpads – a novel memory hierarchy
  - Hides memory layout in hardware
  - Moves objects rather than cache lines
  - Replace caches with pads which store objects efficiently
  - Introduce new instructions which manipulate objects in a safe way
- **Results:** Hotpads outperforms conventional cache hierarchies
  - 34% faster execution
  - 2.6x less energy used
  - Reduced data movement

Background & Problem

Conventional Memory Interface

- Early languages (C, Fortran…) expose memory as a flat address space
- Allow arbitrary loads and stores (unsafe operations)
- Downsides:
  - Invalid pointers (memory corruption)
  - Memory leaks (show up during runtime)
  - Programmers think of objects instead of addresses

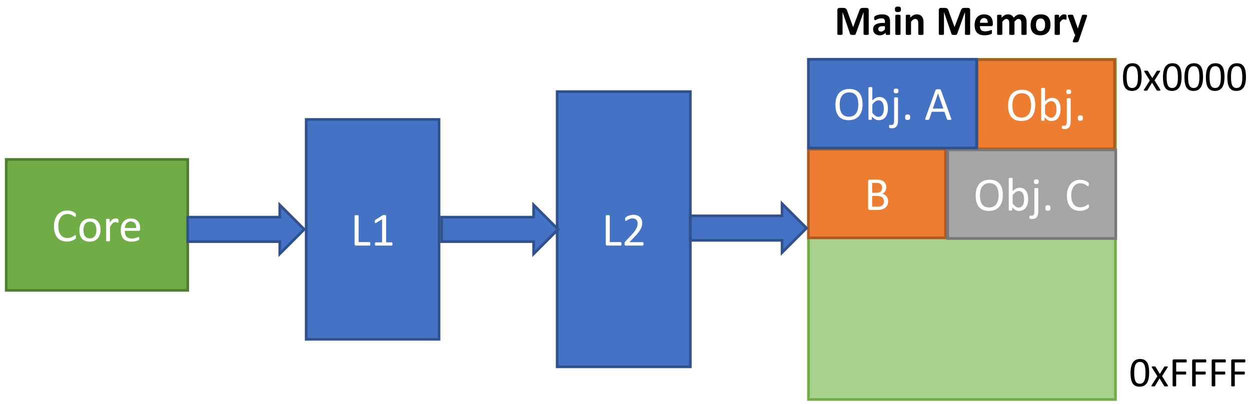


Memory-Safe Languages

- Memory-safe languages strictly **hiding** the flat address space
- Do not expose raw pointers or allow access to arbitrary memory locations
- Provide an **object-based model** to access memory
- Most modern languages (Java, Go, Rust, …) are memory-safe
- Benefits:
  - Objects are more natural for most programmers
  - Memory safety avoids corruption bugs
  - Automatic memory management (garbage collection) simplifies programming
- Downsides:
  - Memory safety adds overhead and performance suffers

Conventional Memory Hierarchy

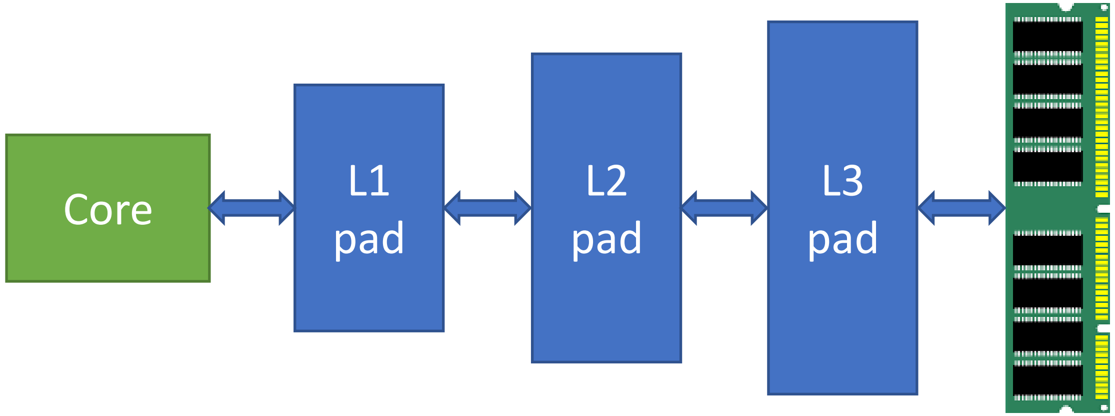
- **Overhead** is caused by a **mismatch** between object-based model and the memory hierarchy exposed as flat address space
- Inefficient as most spatial locality is within an object
- Maintaining the illusion of flat address space requires costly associative lookups



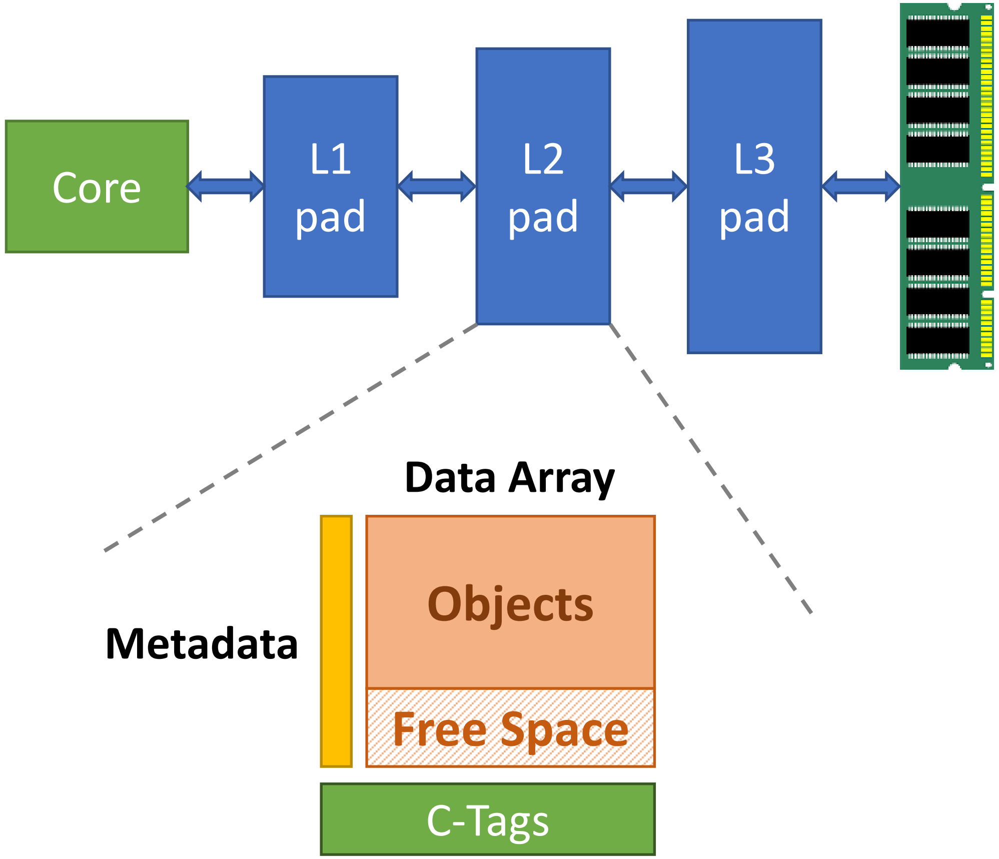
Key Approach & Ideas

Hotpads

- A new memory hierarchy designed from the ground up for object-based programs
- Hotpads uses **pads** which are directly addressed memories managed by hardware
- A pad is maintained using techniques similar to garbage collection
- Hotpads introduces **new instructions** to support pointer operations



Pads



- **Data array**
  - Contiguous region for allocated objects and free space
  - Data array of each pad and the main memory is mapped to different addresses
- **C-Tags**
  - Each object has canonical address which points to *final resting place*
  - C-Tags array maps canonical address to per-level address
- **Metadata**
  - Pointer? Valid? Dirty? Recently-used?

Aside: Scratchpads

- Pads are similar to Scratchpads but consider **semantics** of an object
- Scratchpads are like caches (fast, close to CPU)
- In Scratchpads data has to be **explicitly** moved by programmer
- Pads move data **implicitly** when accessing a memory address

ISA Extension: Overview

- Hotpads ISA treats pointers as **abstract data types** (address may not be accessed)
- Enables the microarchitecture to manipulate pointers safely
- Only one addressing mode `base + offset`
- `base` is always a pointer to the start of an object
- `offset` is an immediate (where is data inside the object?)
- Pointers to **arbitrary locations** within an object are **not allowed**

ISA Extension: Data Load/Store

- The standard load/store instructions can be used to **access non-pointer data** within an object
- `rd` register can only hold data
- `rp` register can only hold pointers
- `disp(rp) = rp + disp` (disp is immediate)
  - `3(0x42)` - *access third word in object which sits at address* `0x42`

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

ISA Extension: Pointer Load/Store

- Load and store instructions to **access pointers within an object**
- Same semantic as before but the system knows that the data accessed is a pointer

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

ISA Extension: Pointer Dereference

- Hotpads includes a **dereference instruction** to facilitate pointer rewriting (only in L1 pad)
- Unlike `ldptr`, `derefptr` indicates that we intend to immediately access the pointed-to object
- Brings pointed-to object into L1 pad

Instruction	Format	Operation

Pointer Dereference	`derefptr rp, disp(rb)`	`rp <- Mem[EffAddr]`
		`brings object in L1`

ISA Extension: Object Allocation

- Hotpads provides an instruction to **allocate a new object**
- Hotpads has a pointer **equality instruction**

Instruction	Format	Operation

Allocation	`alloc rp, size, type`	`NewAddr <- Alloc(size)`
		`Mem[NewAddr] <- type`
		`rp <- NewAddr`

Pointer Equality	`seqptr rd, rp1, rp2`	`rp1 == rp2`

ISA Extension: Objects in Pads

- Objects must be word aligned within a pad (at least two words long)
- The first word contains type id and metadata
- Hotpads tracks **integrity of pad pointers** (cannot transform non-pointer data into a pointer to a pad)
- Relies on language-level memory safety to guarantee **integrity of main memory pointers**

Key Features

Key Features

- Implicit, object-based data movement
- Pointer rewrites
- In-hierarchy object allocation
- Hierarchical garbage collection and evictions

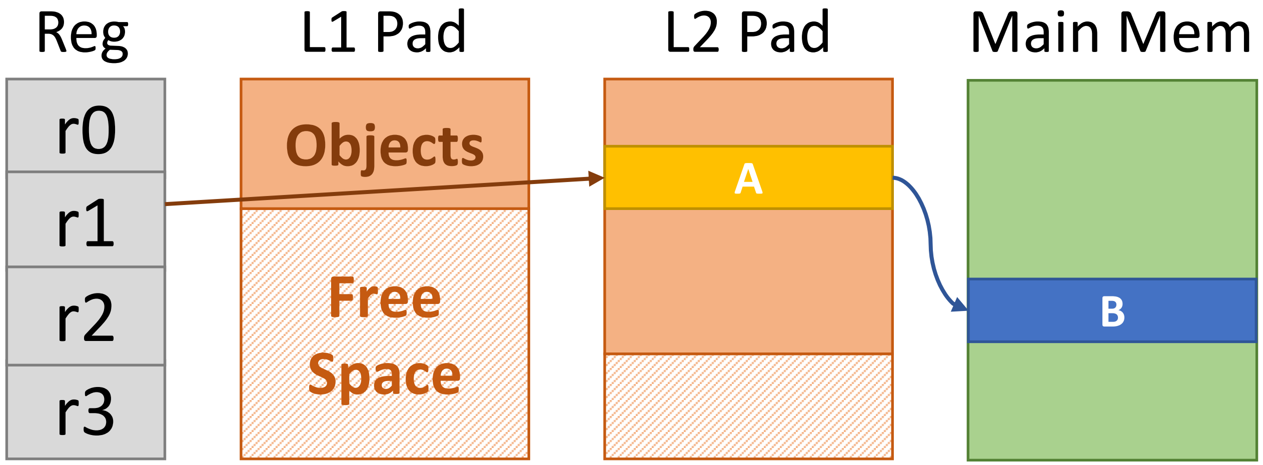
Key Features

- Implicit, object-based data movement
- Pointer rewrites
- In-hierarchy object allocation
- Hierarchical garbage collection and evictions

Data Movement

```
class Node {
   int value;
   Node next;
}
```

```
Node A; Node B;
A.next = B;
```

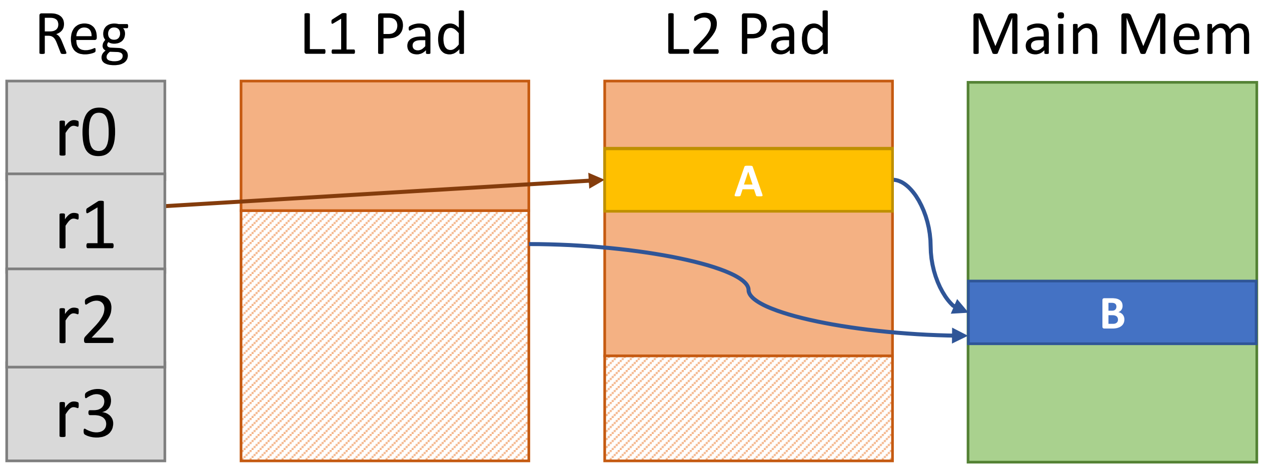


Initial state

Data Movement (con.)

```
class Node {
    int value;
    Node next;
}

int v = A.value;
ld r0, (r1).value
```



A is copied into L1 pad
A in L1 still points to B

- Hotpads moves objects implicitly on access
- Bump

Instruction	Format	Operation
Data Load	`ld rd, disp(rp)`	`rd <- Mem[EffAddr]`

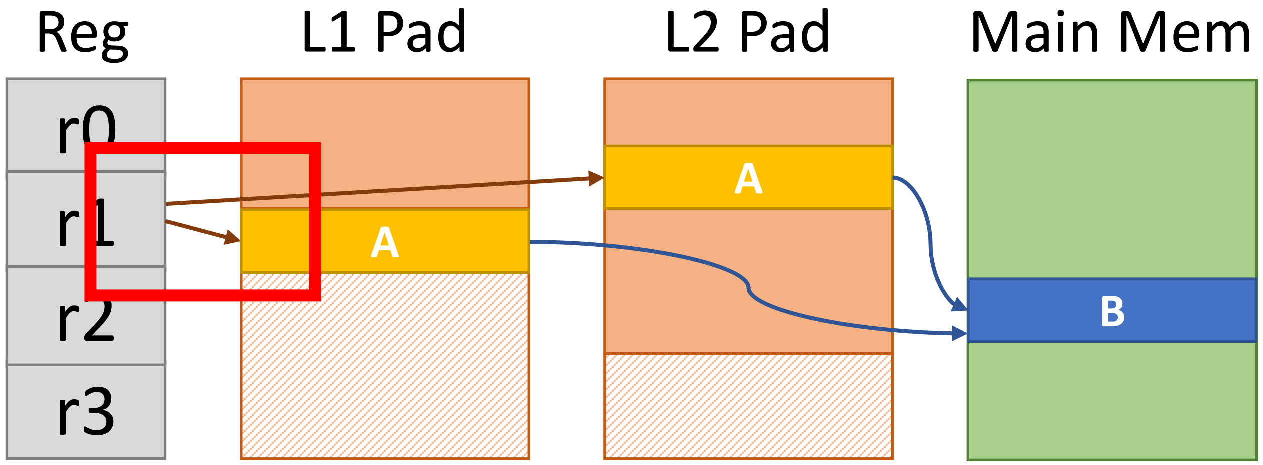
Key Features

- Implicit, object-based data movement
- Pointer rewrites
- In-hierarchy object allocation
- Hierarchical garbage collection and evictions

Pointer Rewrites

```
class Node {
   int value;
   Node next;
}

int v = A.value;
ld r0, (r1).value
```



r1 is rewritten to A's L1 pad address

- Subsequent dereferences of r1 access a copy of A in the L1 directly witho

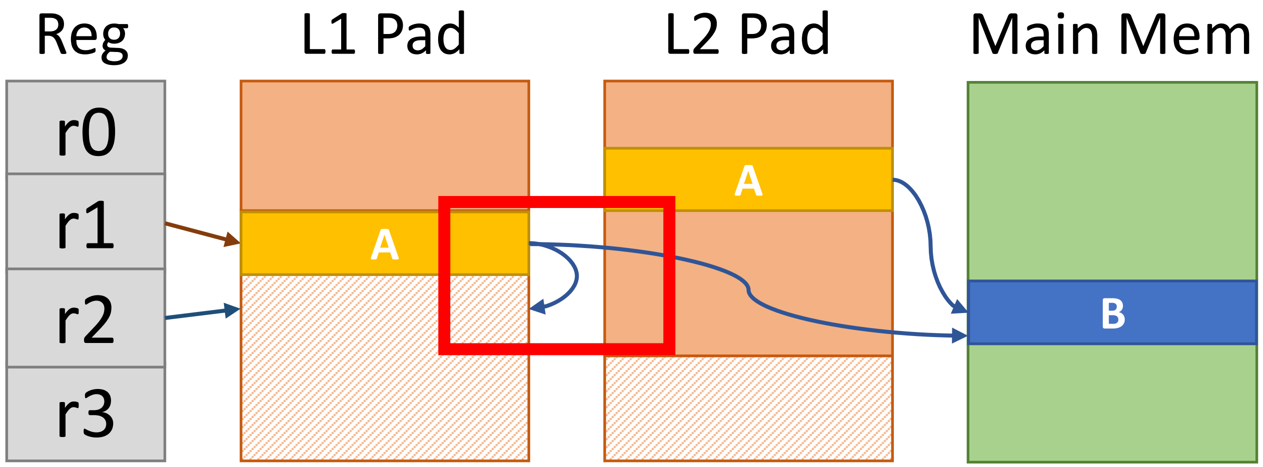
Instruction	Format	Operation
Data Load	`ld rd, disp(rp)`	`rd <- Mem[EffAddr]`

- Hotpads can rewrite pointer safely (memory layout hidden from software)

Pointer Rewrites within Pad

```
class Node {
    int value;
    Node next;
}

int v = A.next.value;
derefptr r2, (r1).next
ld r3, (r2).value
```



B is copied into L1
A’s pointer is rewritten

- Subse rectly withc
- Pads nters

Instruction	Format	Operation
Data Load	`ld rd, disp(rp)`	`rd <- Mem[EffAddr]`

Pointer Dereference	`derefptr rp, disp(rb)`	`rp <- Mem[EffAddr]`
		`brings object in L1`

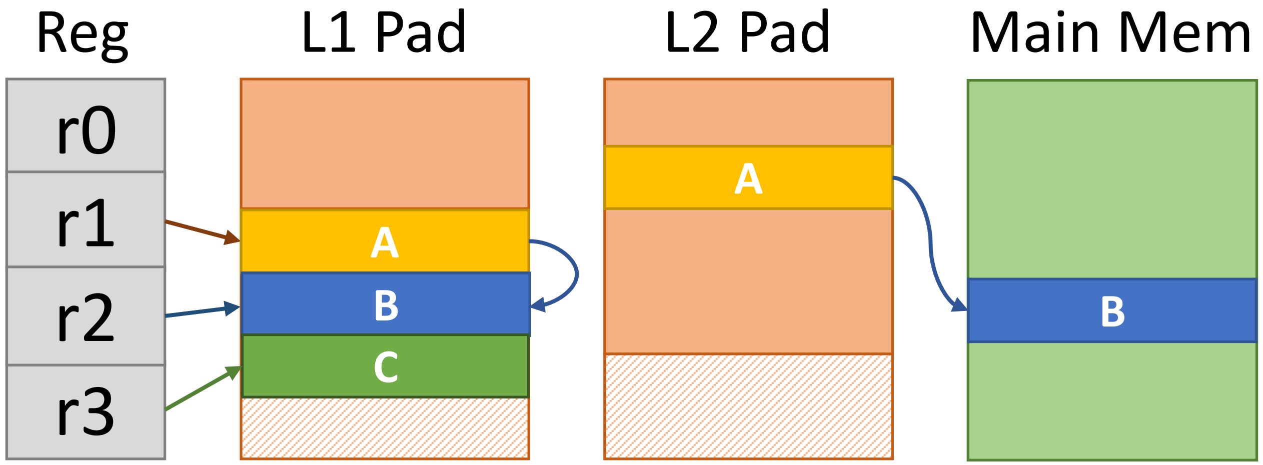
Key Features

- Implicit, object-based data movement
- Pointer rewrites
- In-hierarchy object allocation
- Hierarchical garbage collection and evictions

Object Allocation

```
class Node {
    int value;
    Node next;
}

Node C = new Node();
alloc r3, type=Node
```



CPU allocates new object C

- Requir
- Node (
- In-hier

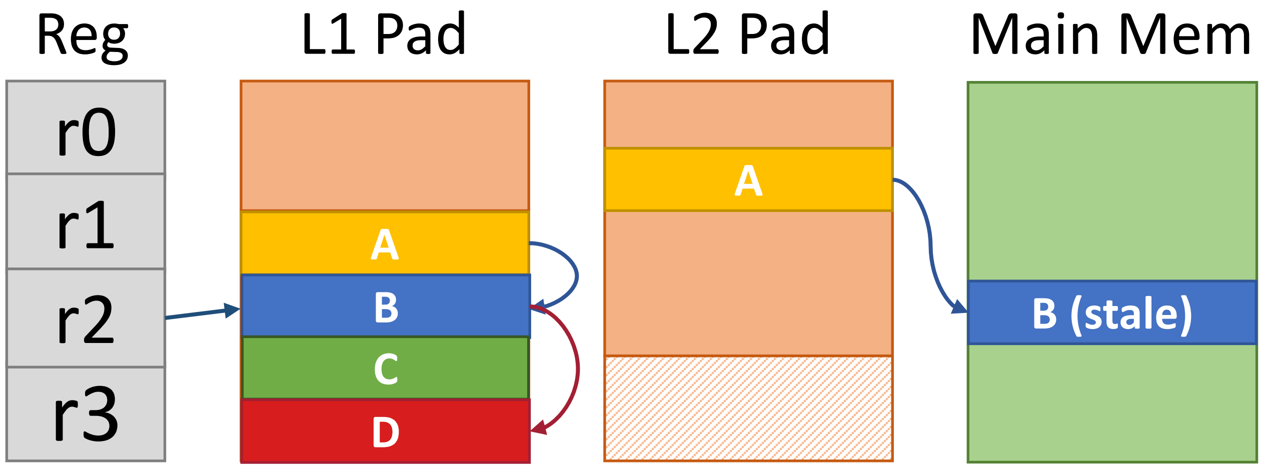
Instruction	Format	Operation

Allocation	`alloc rp, size, type`	`NewAddr <- Alloc(size)`
		`Mem[NewAddr] <- type`
		`rp <- NewAddr`

Key Features

- Implicit, object-based data movement
- Pointer rewrites
- In-hierarchy object allocation
- Hierarchical garbage collection and evictions

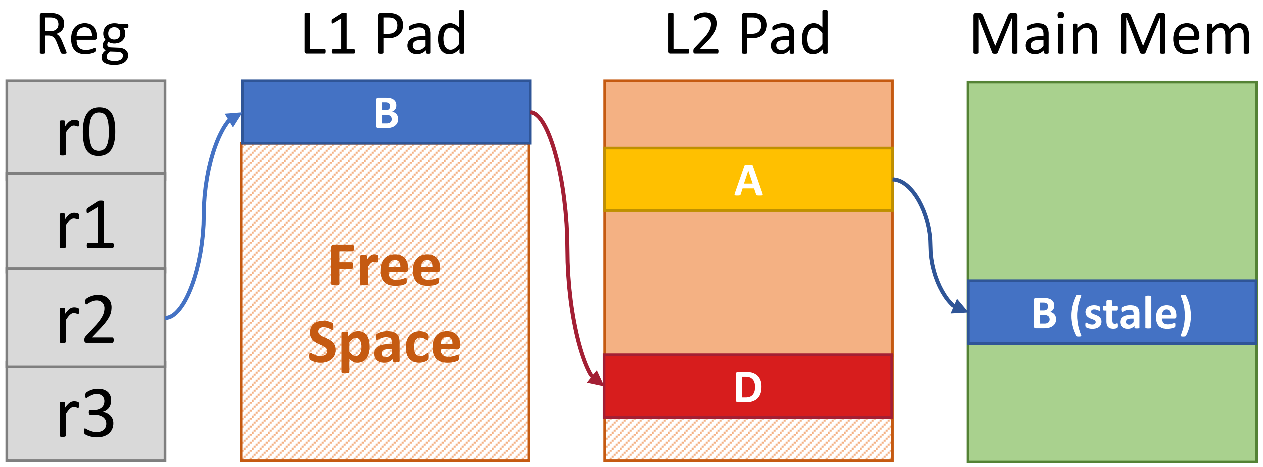
Garbage Collection and Evictions



L1 pad is full

- When a pad fills up, it triggers a collection-eviction (CE) to free space
- Removes dead objects
- Evicts live, not-recently used objects to the next-level pad
- **C** is dead (unreferenced). Other objects are live. Only **B** is recently used.

Garbage Collection and Evictions (con.)



L1 collection-eviction (CE) collects dead C and evicts live A & D to L2

- CEs happen concurrently with program execution
- Each pad can perform a CE independently from higher-level pads
    - Makes CE cost proportional to pad size
    - Here: no need to check L2 pad when performing collection-eviction in L1 pad

Key Features

- Implicit, object-based data movement
- Pointer rewrites
- In-hierarchy object allocation
- Hierarchical garbage collection and evictions

Additional Features

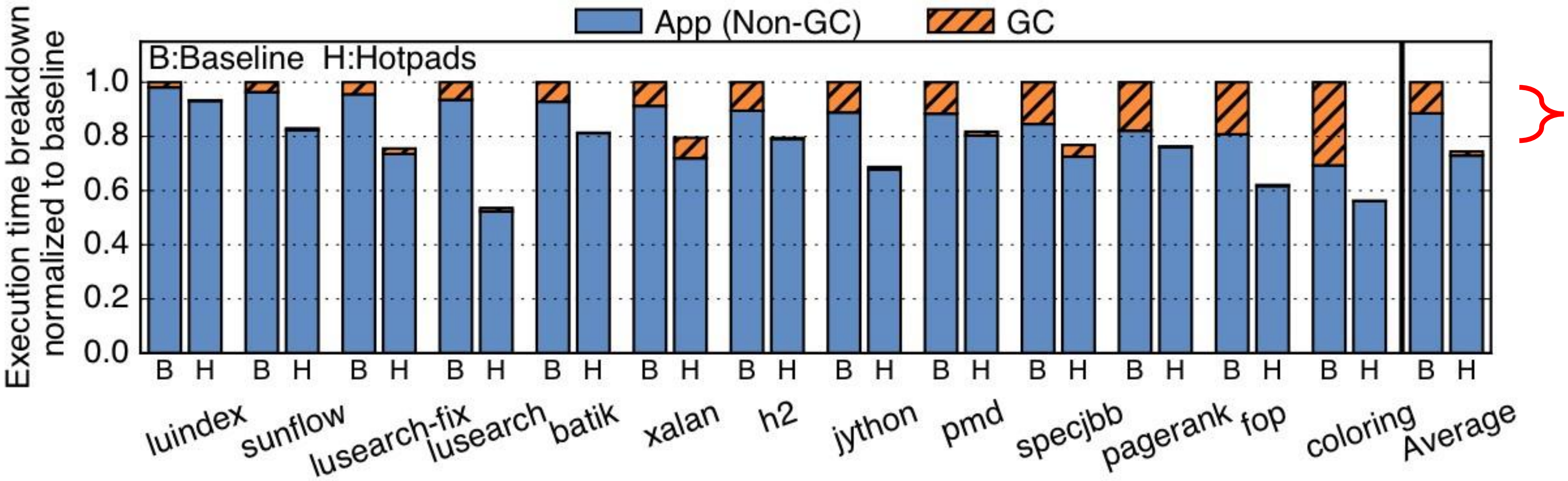
- Support for large objects which do not fit in particular pad
  - Objects can be split in subobjects
  - Subobjects use pads like caches
- Object-level coherence
  - Modified version of MESI cache-coherence protocol
  - Support for multi-core processors
- Legacy mode for flat-address-based programs
  - Uses one large object for all their memory
  - 4% slower than traditional caches on average (up to 14%)

Evaluation & Results

Methodology

- Simulation with MaxSim
  - ZSim (for architecture simulation)
  - Maxine JVM (modified to use Hotpads ISA)
- 4-core processor with out-of-order execution enabled
- Caches: L1, L2, L3 (shared)
- Pads: L1D, L1I, L2, L3 (shared)
- Workloads
  - 13 Java workloads from Dacapo, SpecJBB and JgraphT

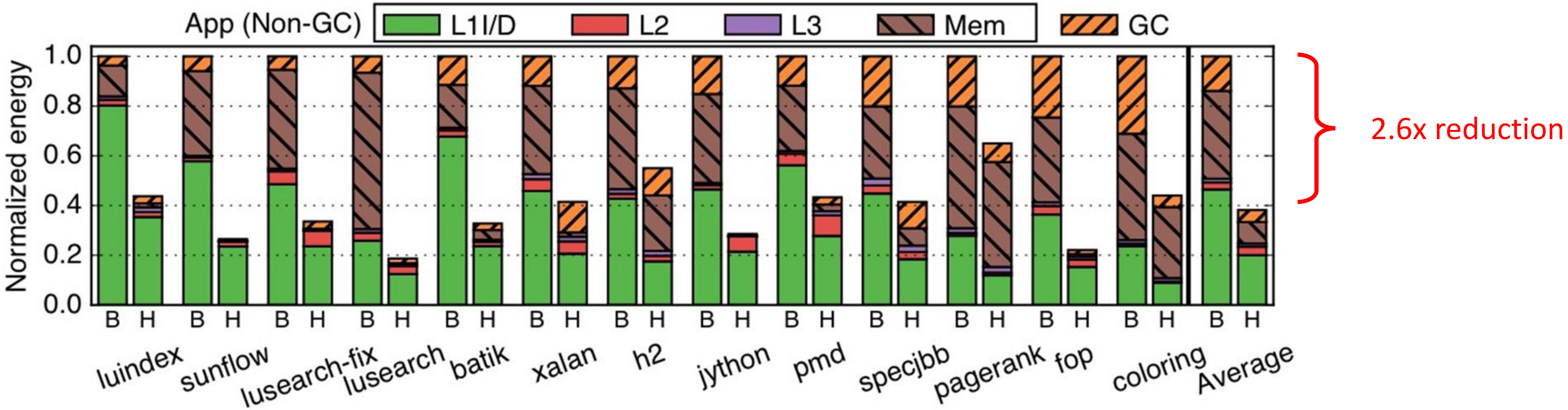
Execution Time Improvement



34% improvement

- Hotpads outperforms conventional cache hierarchies
- In-hierarchy allocation reduces memory congestion
  - App moves less data around which saves time
- Hardware-based collection-evictions reduce GC overheads
  - Less time is spent doing garbage collection

Dynamic Energy Savings

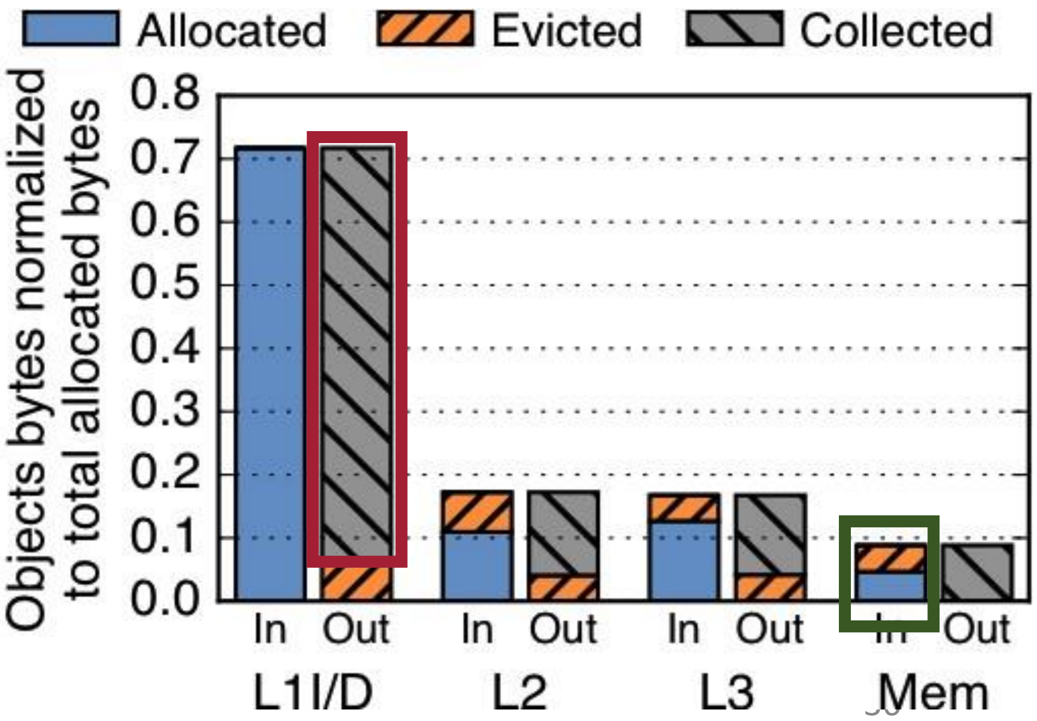


2.6x reduction

- Hotpads reduces dynamic energy consumption in memory hierarchy
- Pointer rewrites enable direct access to L1 data
  - App uses less energy to access frequent data
- Hierarchical collection-eviction collects objects early
  - Less energy is used in main memory and during garbage collection

Data Movement Benefits

- Hotpads reduces data movement
  - Most objects are collected in L1 pad
  - 90% of objects never reach main memory
- Hotpads unifies the locality-principle and the generational hypothesis
  - Eviction keeps recently used objects close to CPU
  - Most objects get collected before reaching higher-levels



Executive Summary

- **Problem:** flat address space is inefficient for memory-safe languages
  - Move cache lines instead of objects (ignore semantics of objects)
  - Short-lived objects require backing storage in main memory
- **Solution:** Hotpads – a novel memory hierarchy
  - Hides memory layout in hardware
  - Moves objects rather than cache lines
  - Replace caches with pads which store objects efficiently
  - Introduce new instructions which manipulate objects in a safe way
- **Results:** Hotpads outperforms conventional cache hierarchies
  - 34% faster execution
  - 2.6x less energy used
  - Reduced data movement

Critique

Strengths of the Paper

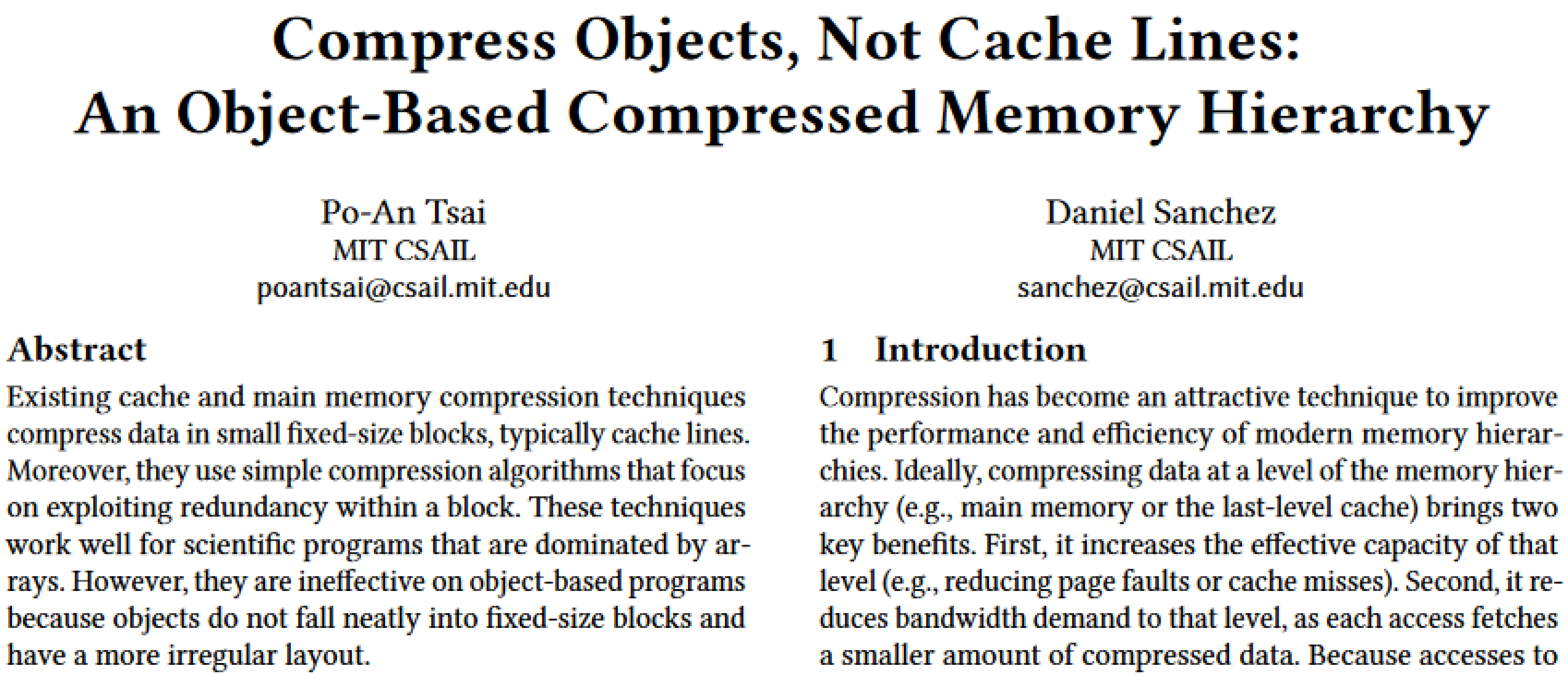
- Correctly identified performance bottleneck for memory-safe languages
- Greatly improves performance and efficiency
- Legacy programs are still supported (but slower)
- Simulated on a multi-core processor

Weaknesses of the Paper

- Hotpads requires a lot of changes in the memory hierarchy
- New hardware required for pads
- Flat-address space is often a good abstraction in scientific computing (e.g. Matrix-Matrix-Multiplication) were performance really matters
- Pointer rewrites only in L1 pad possible
- Many concepts are mentioned again and again but not much details
- Only high-level description of microarchitecture
- No information about virtual addresses or multiple processes

Follow-Up Work

- Extending Hotpads with compression



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.

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

Thoughts & Takeaways

- It’s the right time to redesign the memory hierarchy because there is a trend towards specialized hardware
- Take a fresh look at “the way we’ve always done things” and do better
- Transfer existing concepts from software to hardware get benefits for free
- Observe current trends and adapt hardware to make programmers life easier

Questions?

Discussion

- Do you think the current memory hierarchy can be modified in such a drastic manner?
- Do you have other ideas on how to improve the memory hierarchy for modern languages?
- Will it be easy to implement the microarchitecture (pointer rewrites, CE…) with existing technologies?
- As Hotpads hides memory addresses, do you think it can be more secure than existing memory hierarchies (e.g., Spectre, Meltdown?)