Architectures and Programming Models for General-Purpose Near-Data Computing

Brian C. Schwedock

Memory-Centric Computing Systems @ MICRO'24

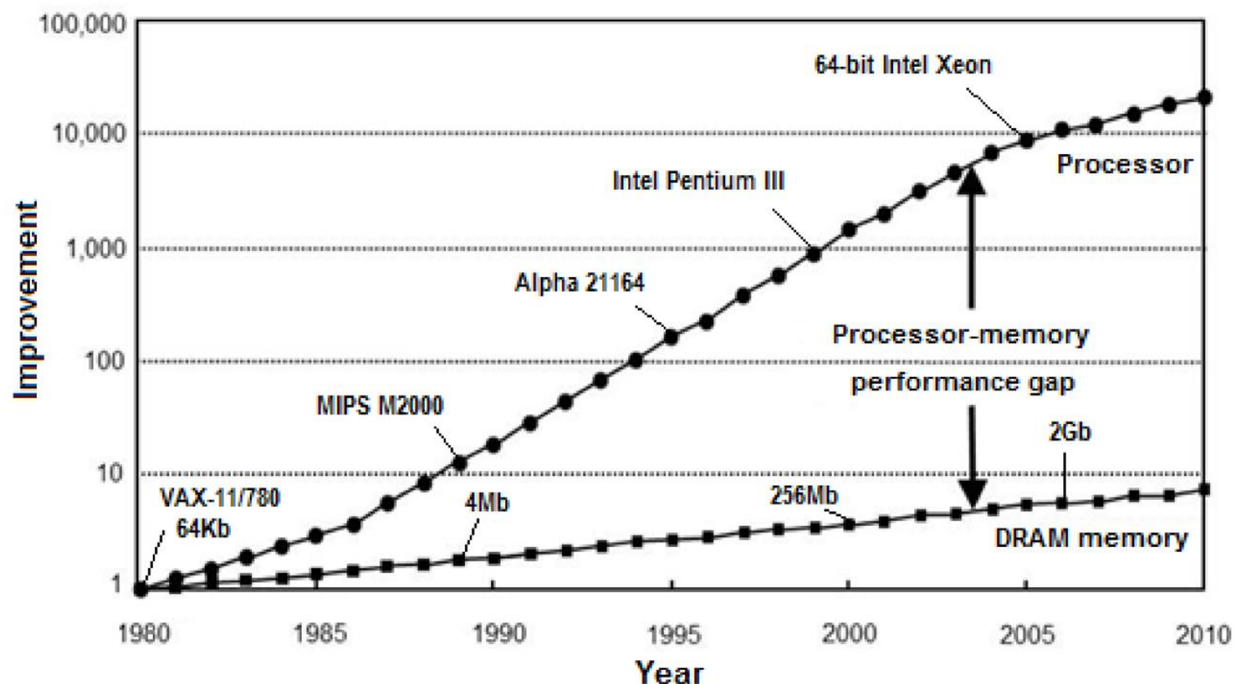
November 2, 2024

Carnegie Mellon University

SAMSUNG

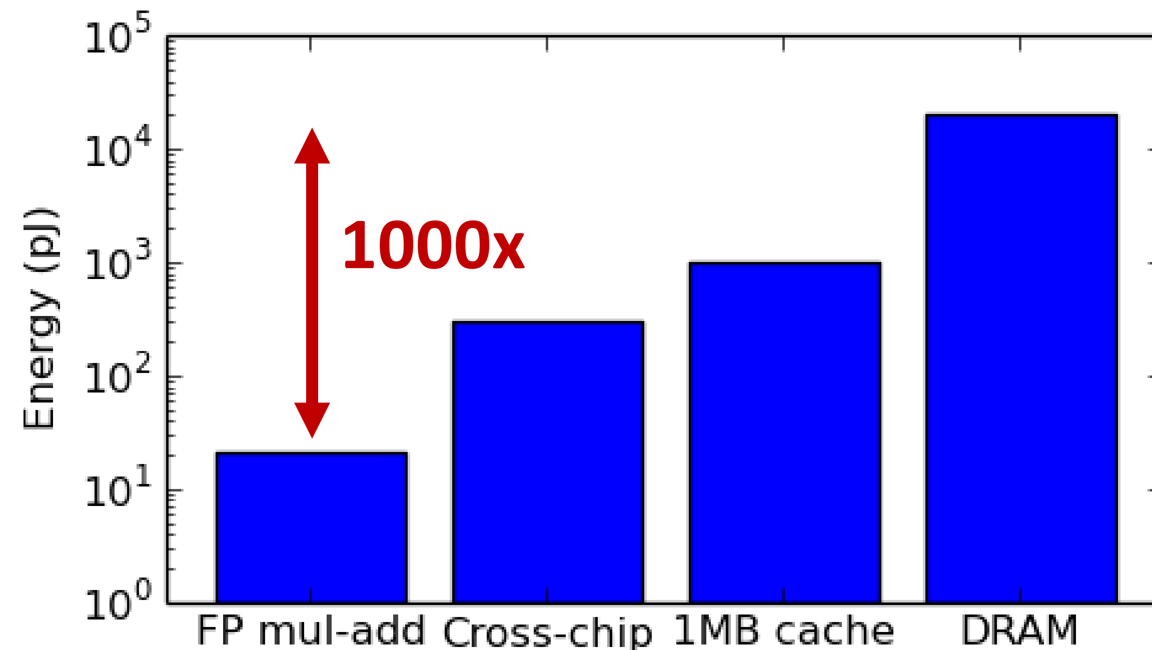
Data movement costs keep getting worse

Performance



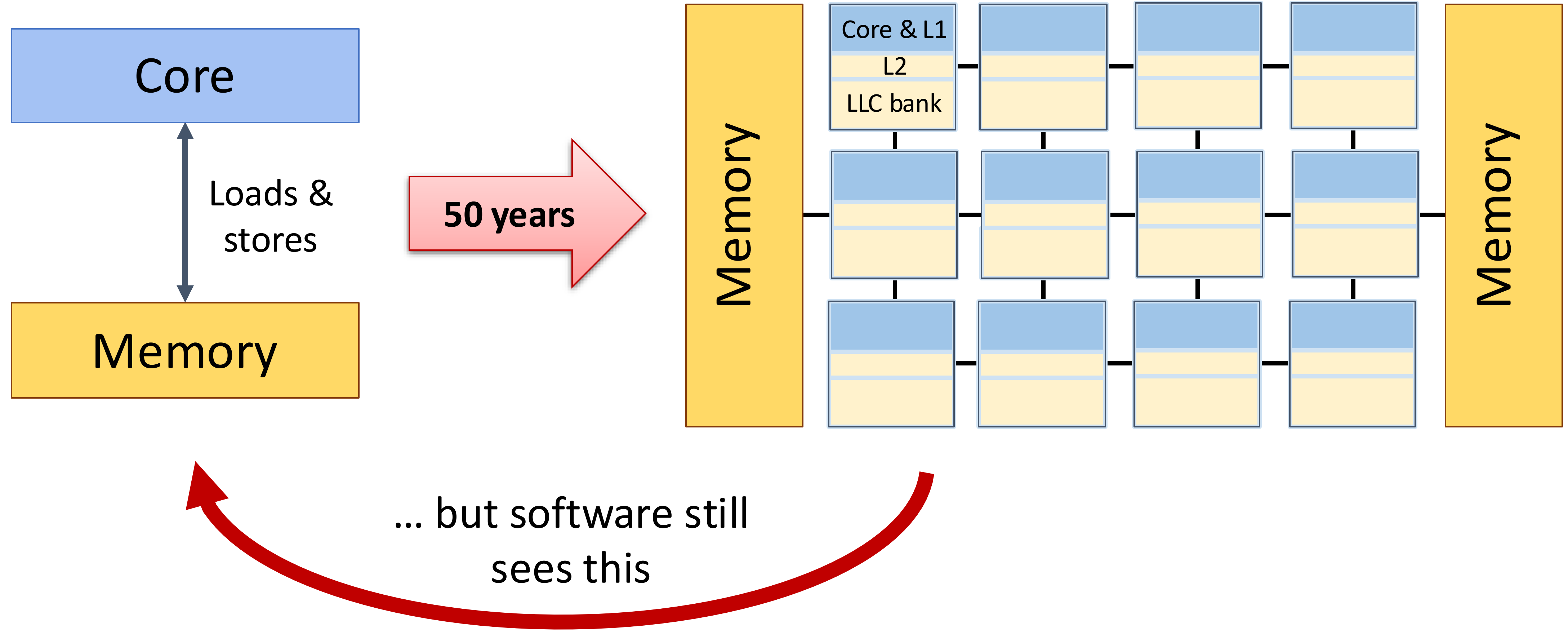
[Efnusheva, IJCSIT'17]

Energy

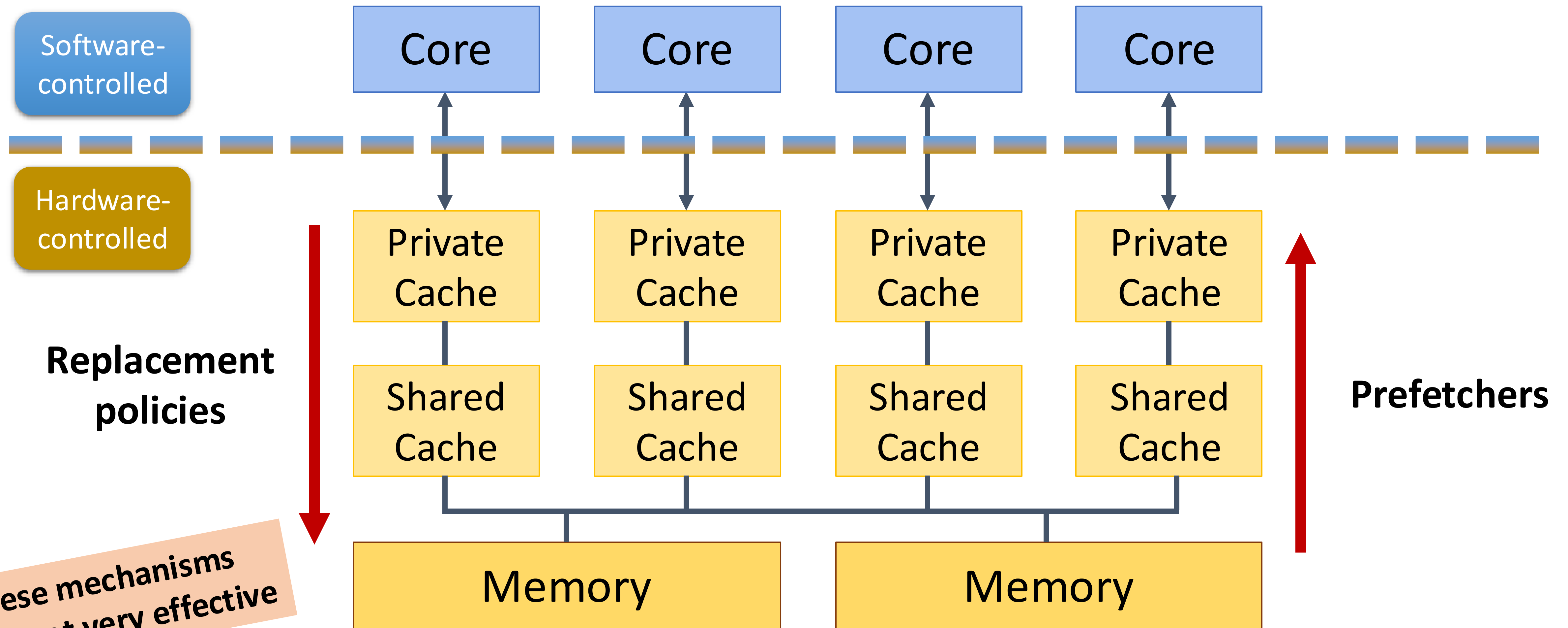


[Dally, SC'10]

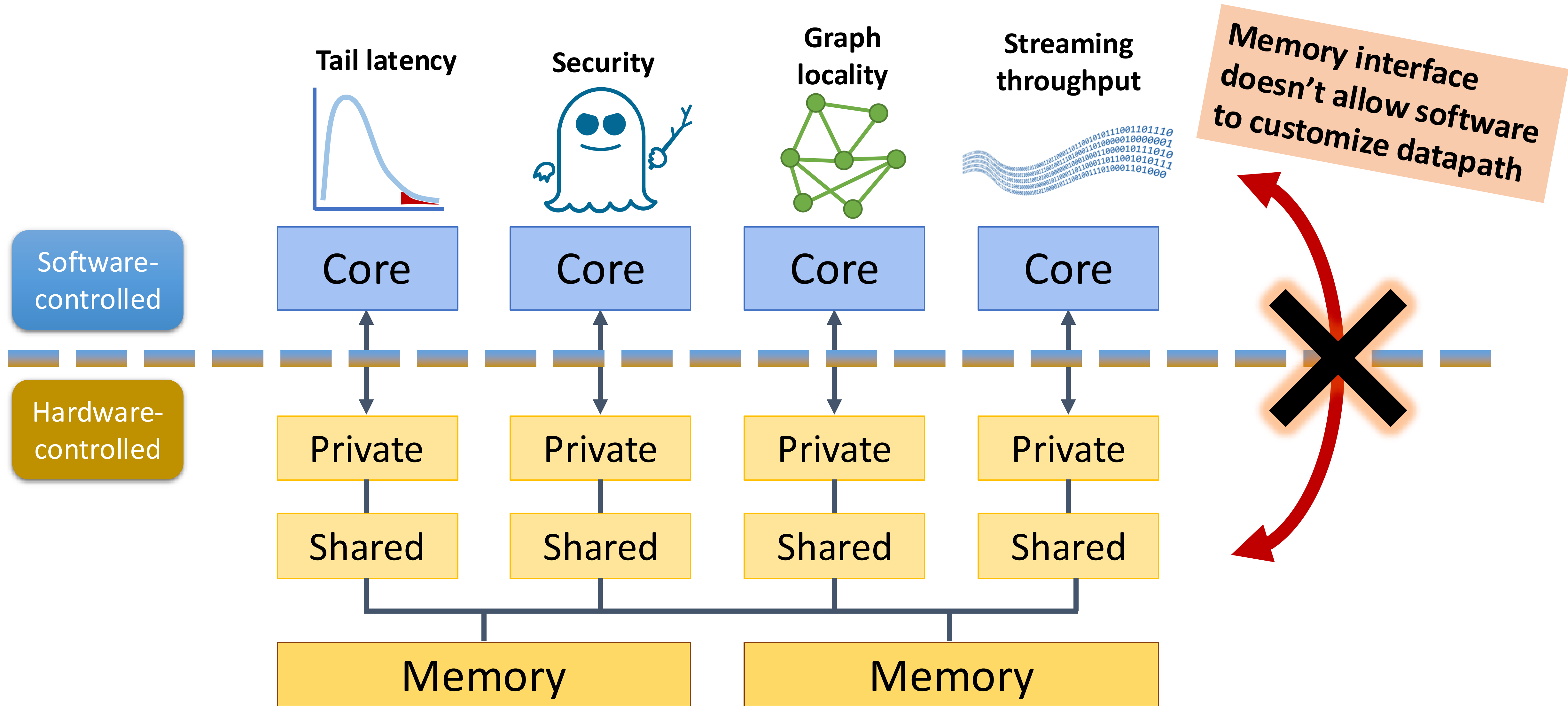
Memory hierarchy has evolved, but the interface has not



Data movement optimizations are forced to be conservative



Hardware doesn't know what applications want



Specialized memory hierarchies (e.g., NDC, PIM) give big gains!

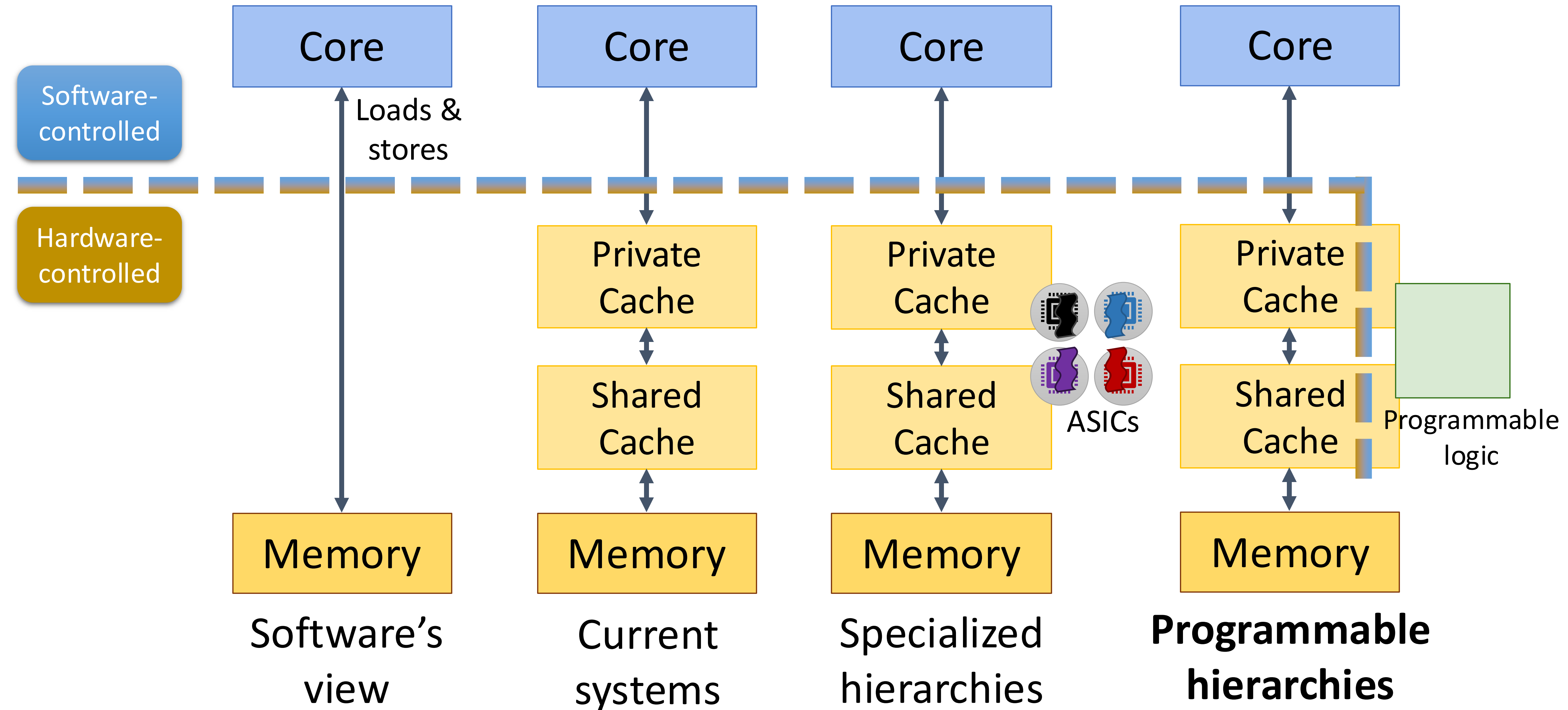
Graph optimizations
Security
Data placement
Remote memory operations
Sparse linear algebra / machine learning
Approximation / compression
Data marshalling / layout
Pointer chasing
Memoization
Prefetching
Relaxed coherence / consistency
Memory management
…

- Big opportunity -

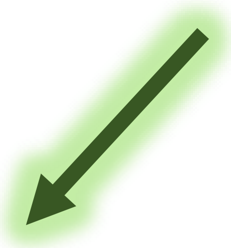
Often >2x end-to-end speedup!

How can we specialize the memory hierarchy without specialized hardware?

The *interface* is the problem!

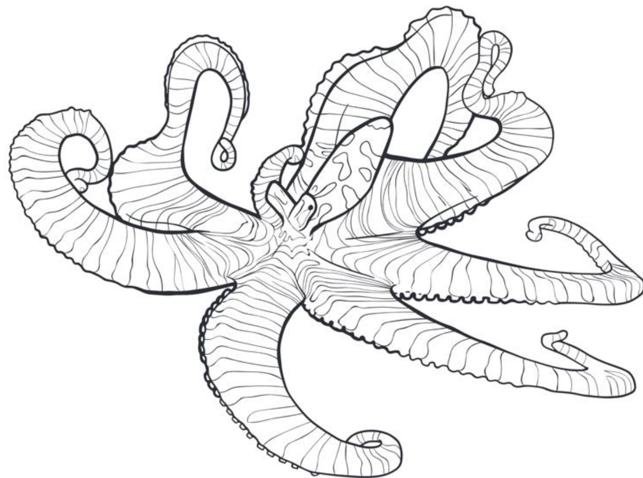


Overview



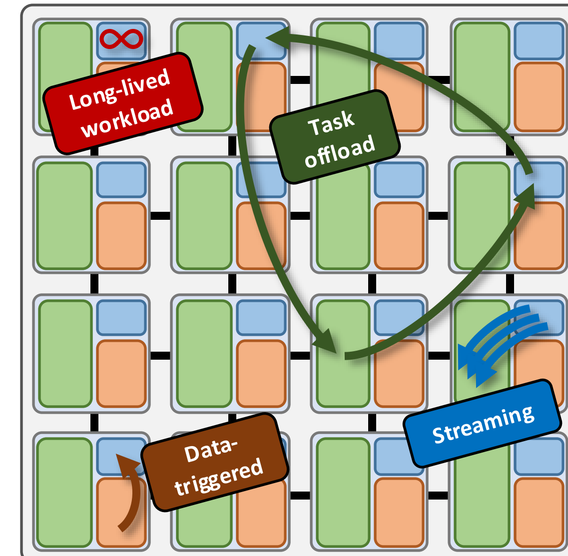
täkō

ISCA 2022



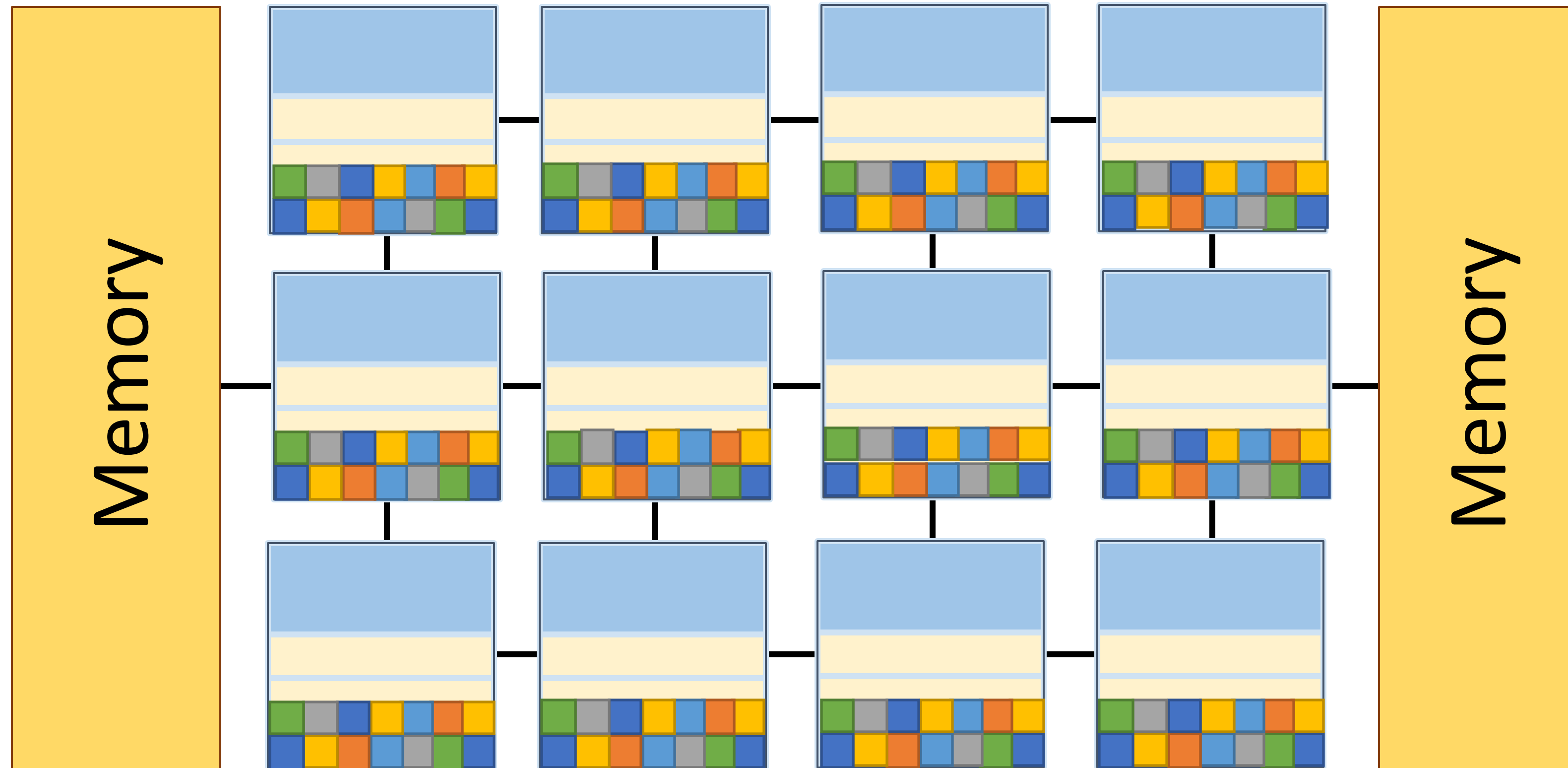
Leviathan

MICRO 2024

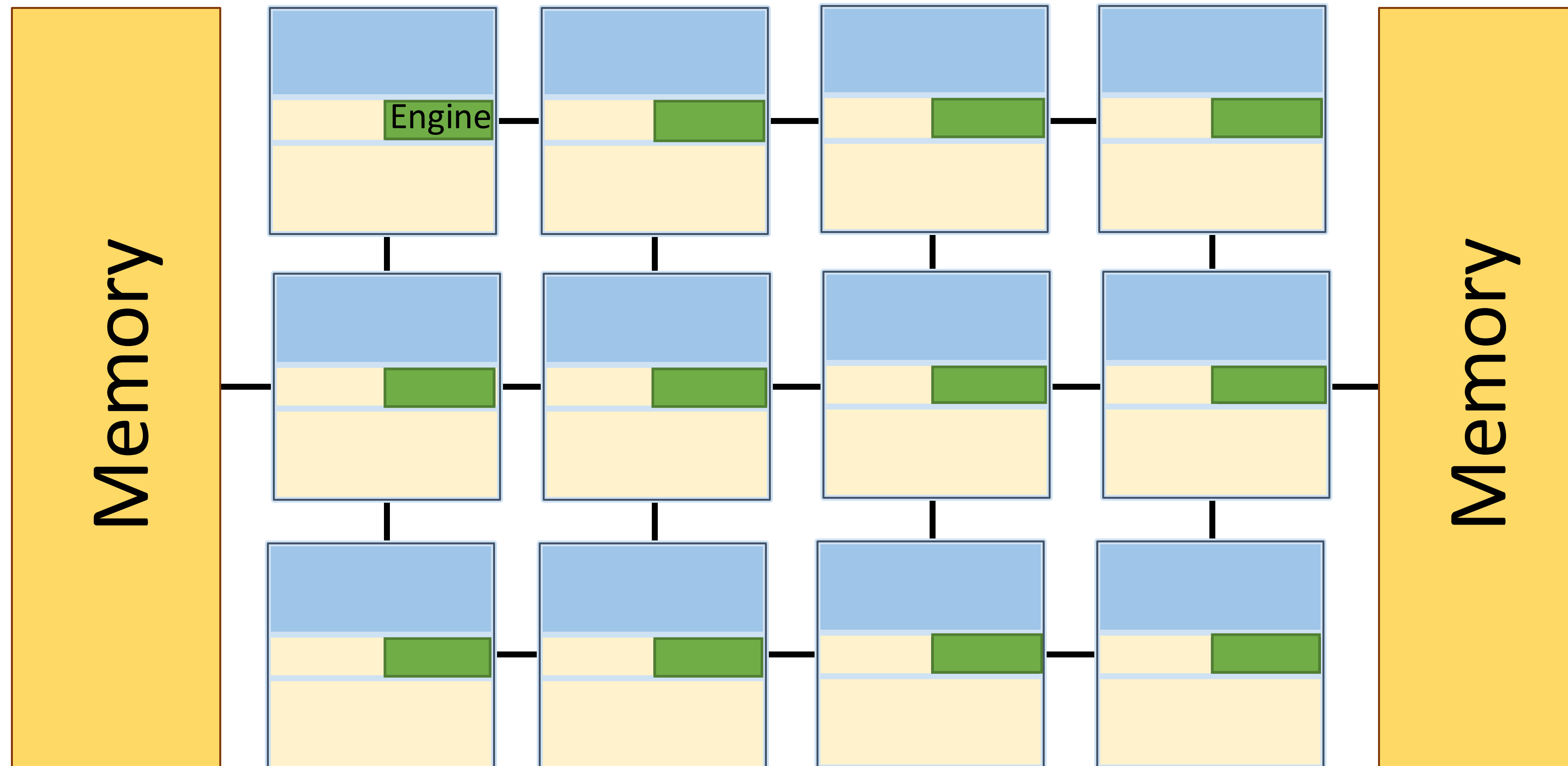


* täkō is Japanese for “octopus”

Programmability replaces the need for custom hardware



Programmability replaces the need for custom hardware



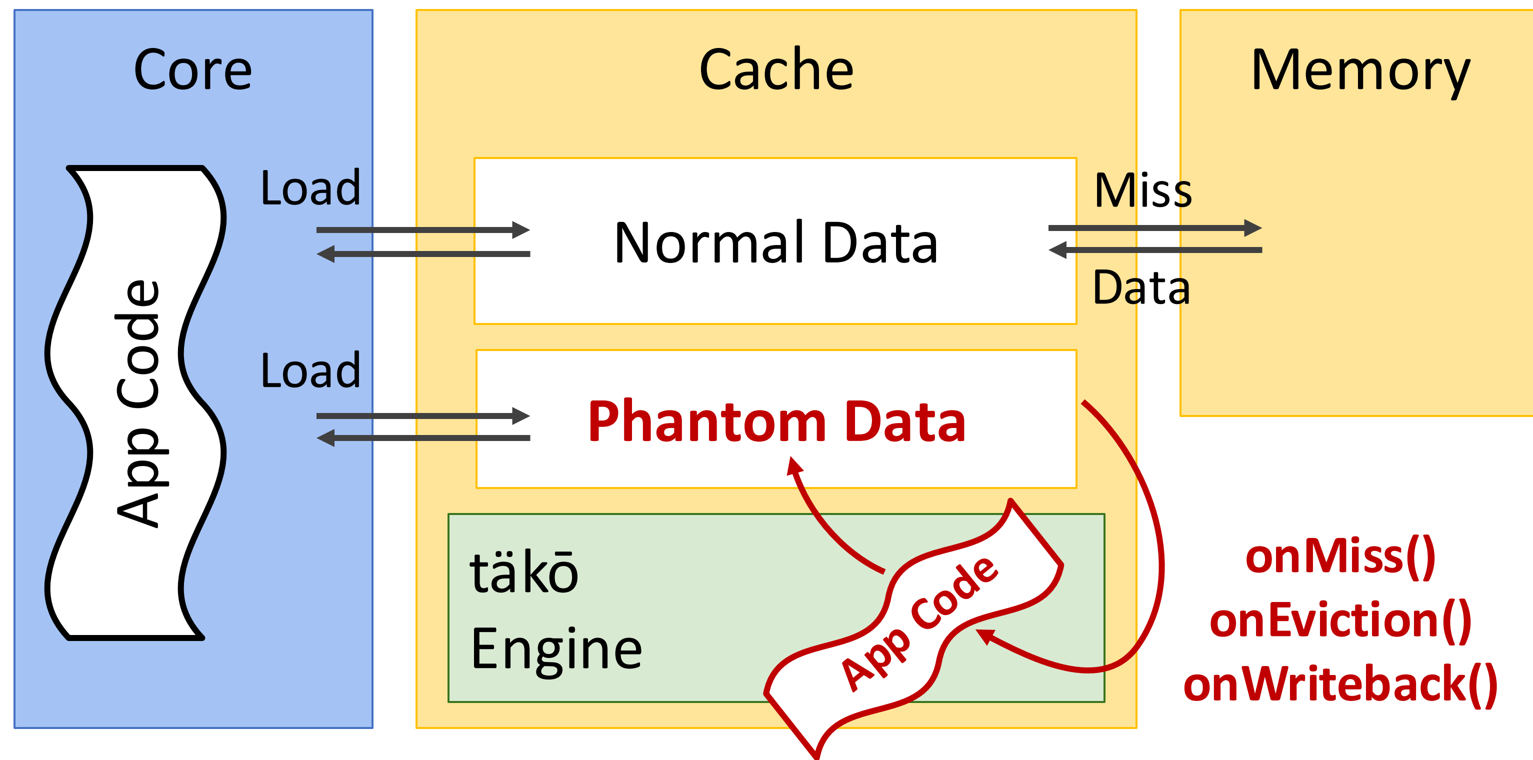
täkō exposes data movement to software

Solution: Tell software when data moves! (& let it do something)

Baseline loads & stores work normally

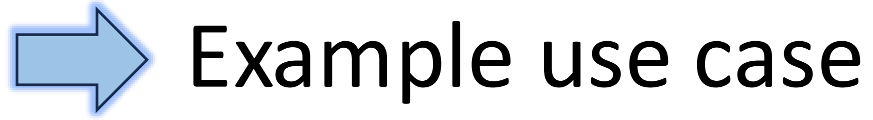
"Phantom" data lives only in caches! *[Carter, HPCA'99]*

Software serves misses to phantom data (& evictions)



täkō: Roadmap

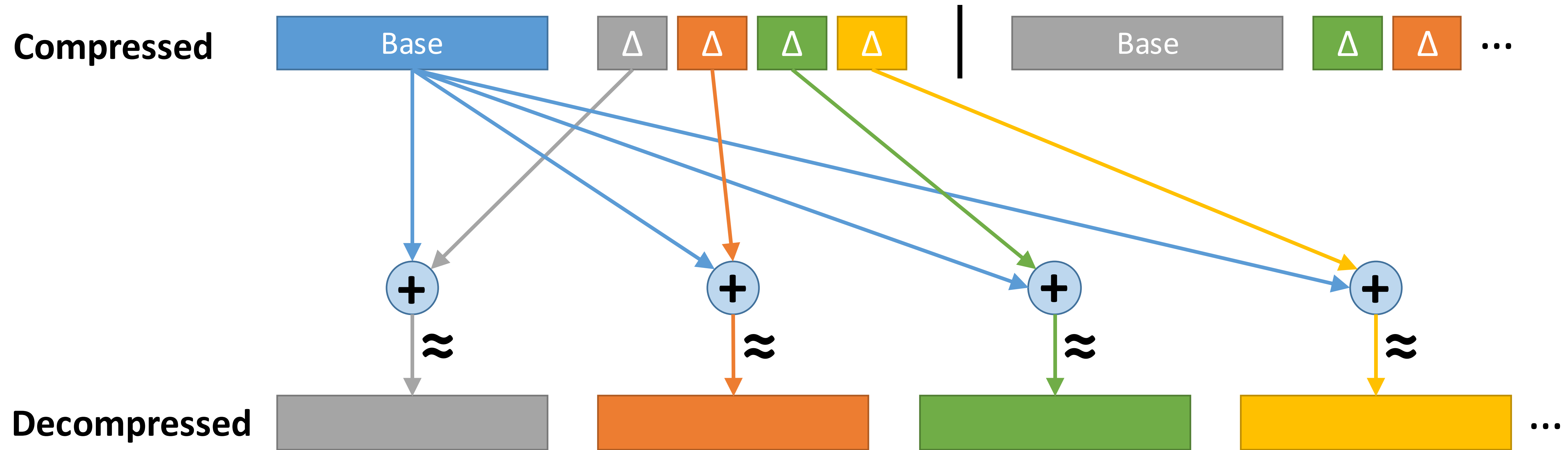
Introduction



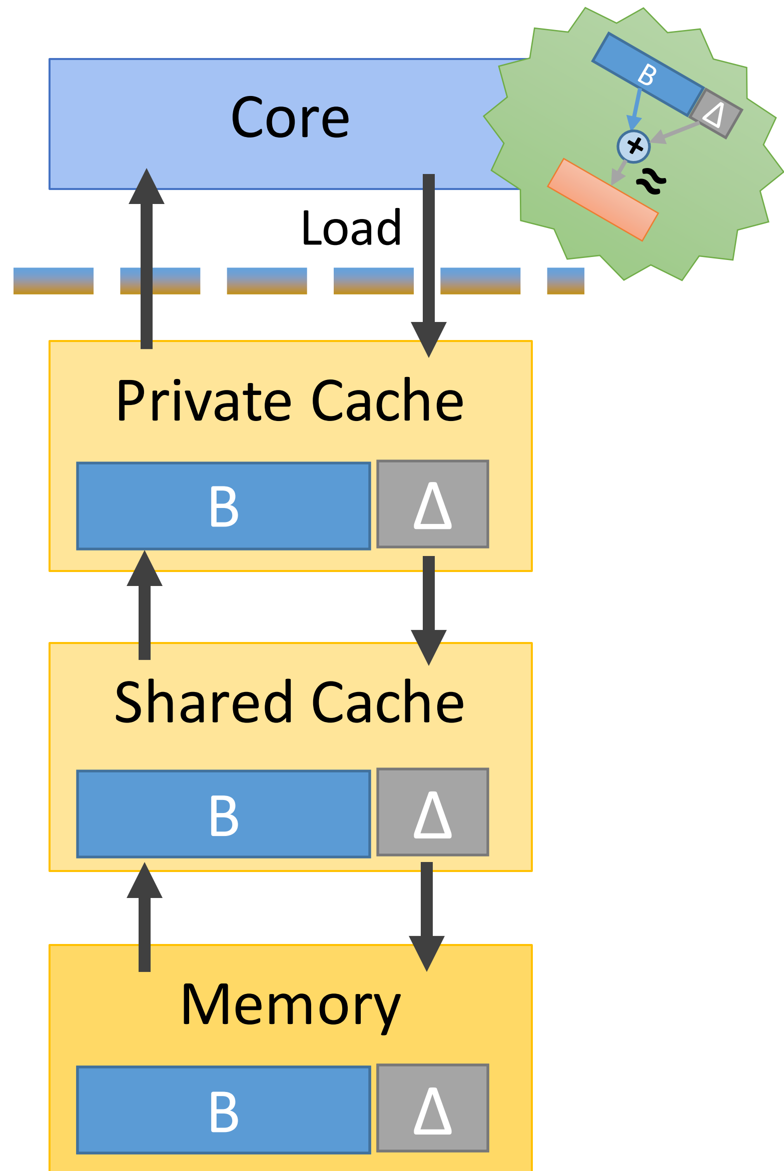
täkō design

Evaluation

Base+delta (B+Δ) **lossy** compression



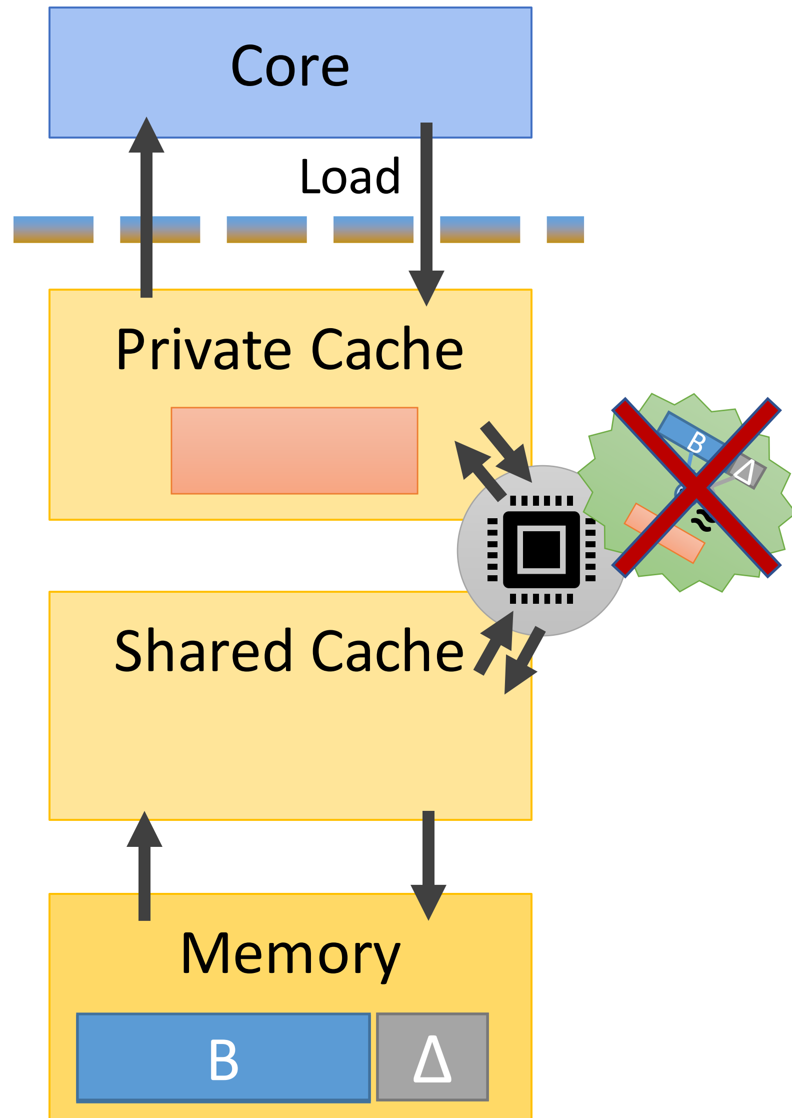
B+Δ decompression: in the core



Pros: Reduced memory and cache usage

Cons: Decompress on every access

B+Δ decompression: compressed caches?



Pros:

Reduced memory and cache usage

Reuse decomp. data in private cache

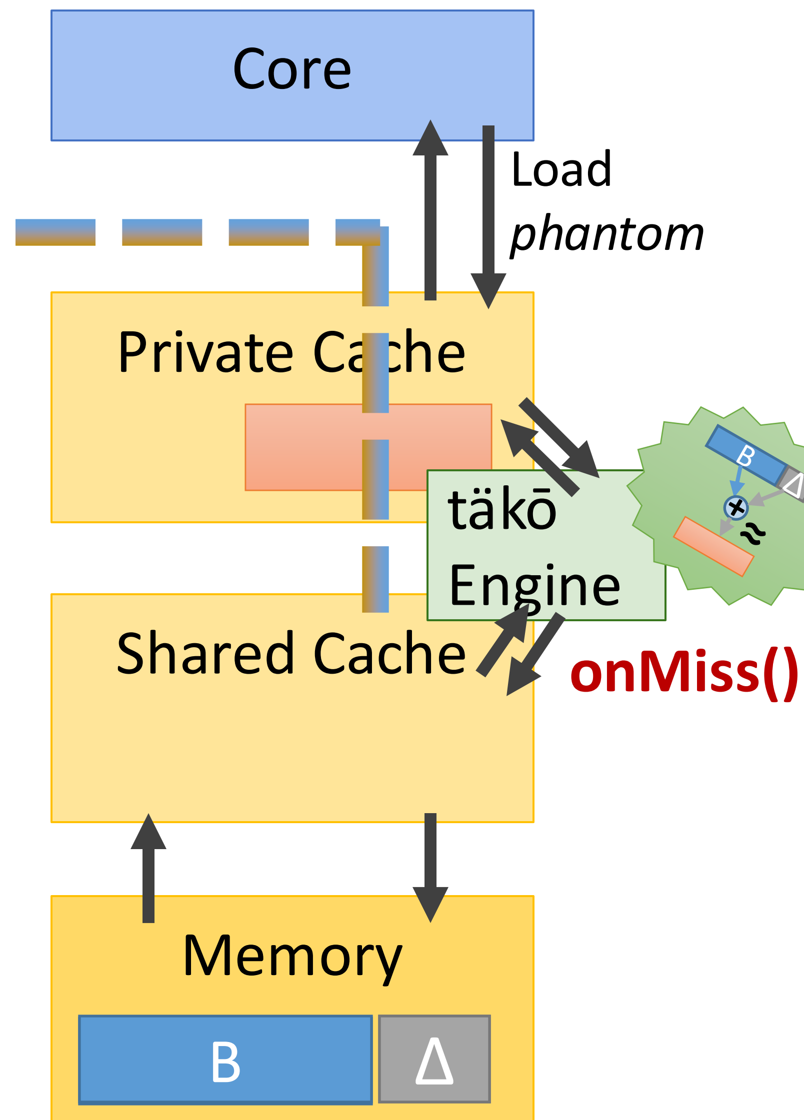
Cons:

Compression scheme fixed in hardware

Cannot support lossy compression

[Miguel, MICRO'15]

B+Δ decompression: täkō



Pros:

Reduced memory and cache usage

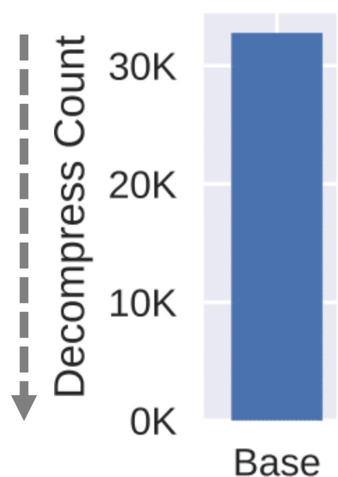
Reuse decomp. data in private cache

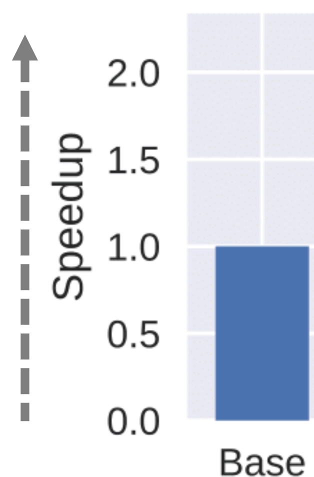
SW can implement any compression scheme!

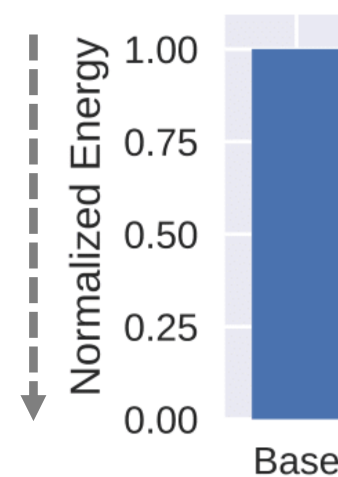
… without needing new hardware for each!

Evaluation: täkō leverages cache to capture reuse

16K objects, 32K accesses, Zipfian distribution (full experimental methodology shown later)



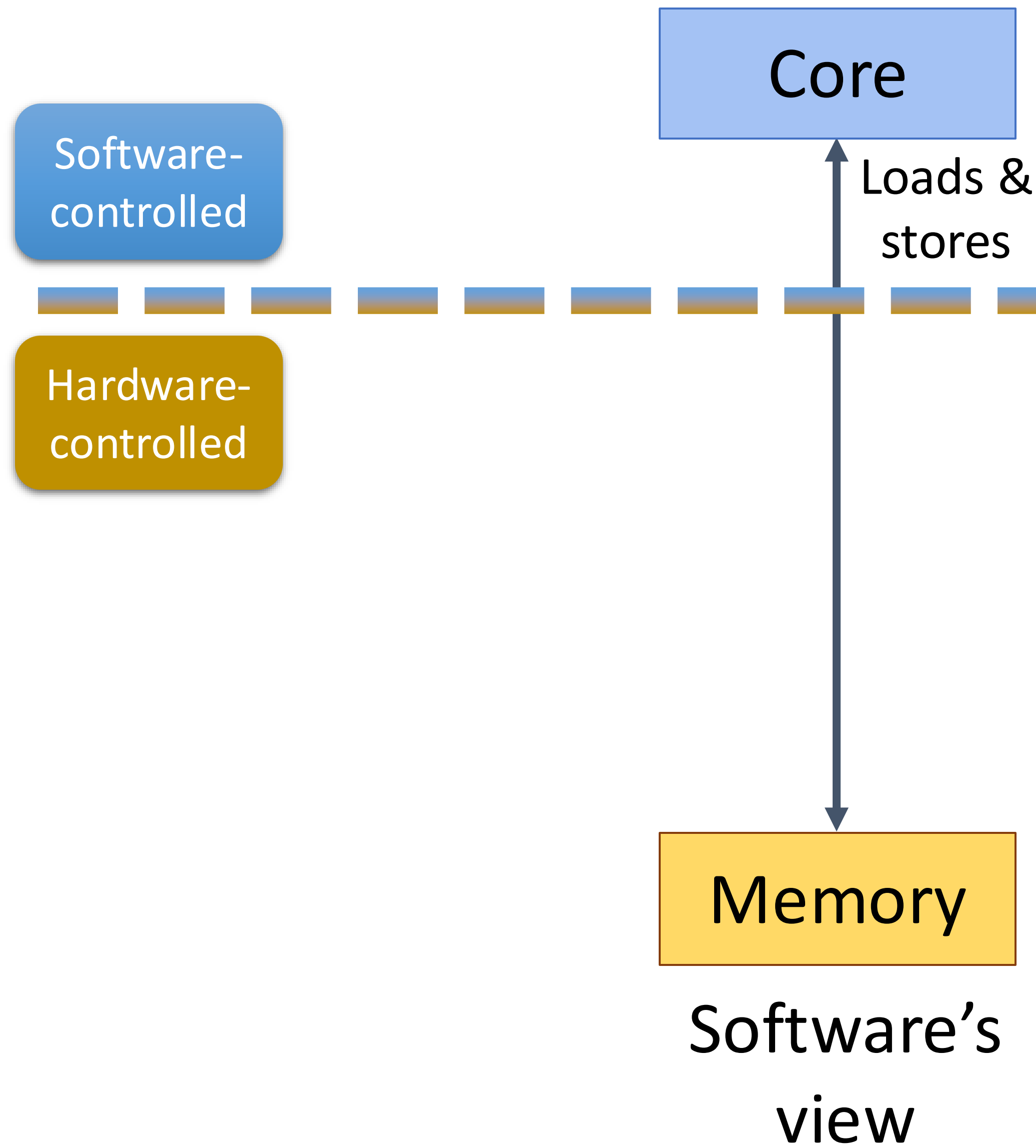


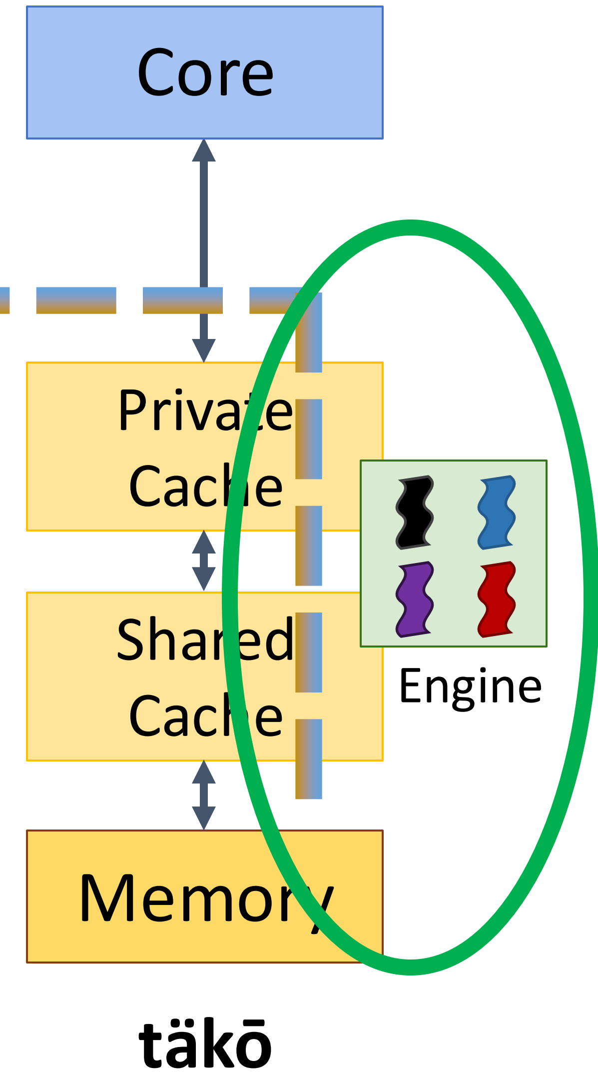


[illegible] n to L2
decompression on cache miss!

The interface is the problem

täkō is the solution

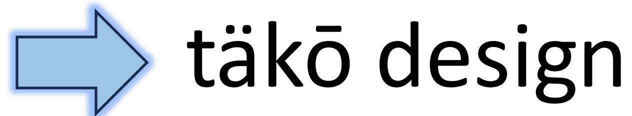




täkō: Roadmap

Introduction

Example use case



täkō design

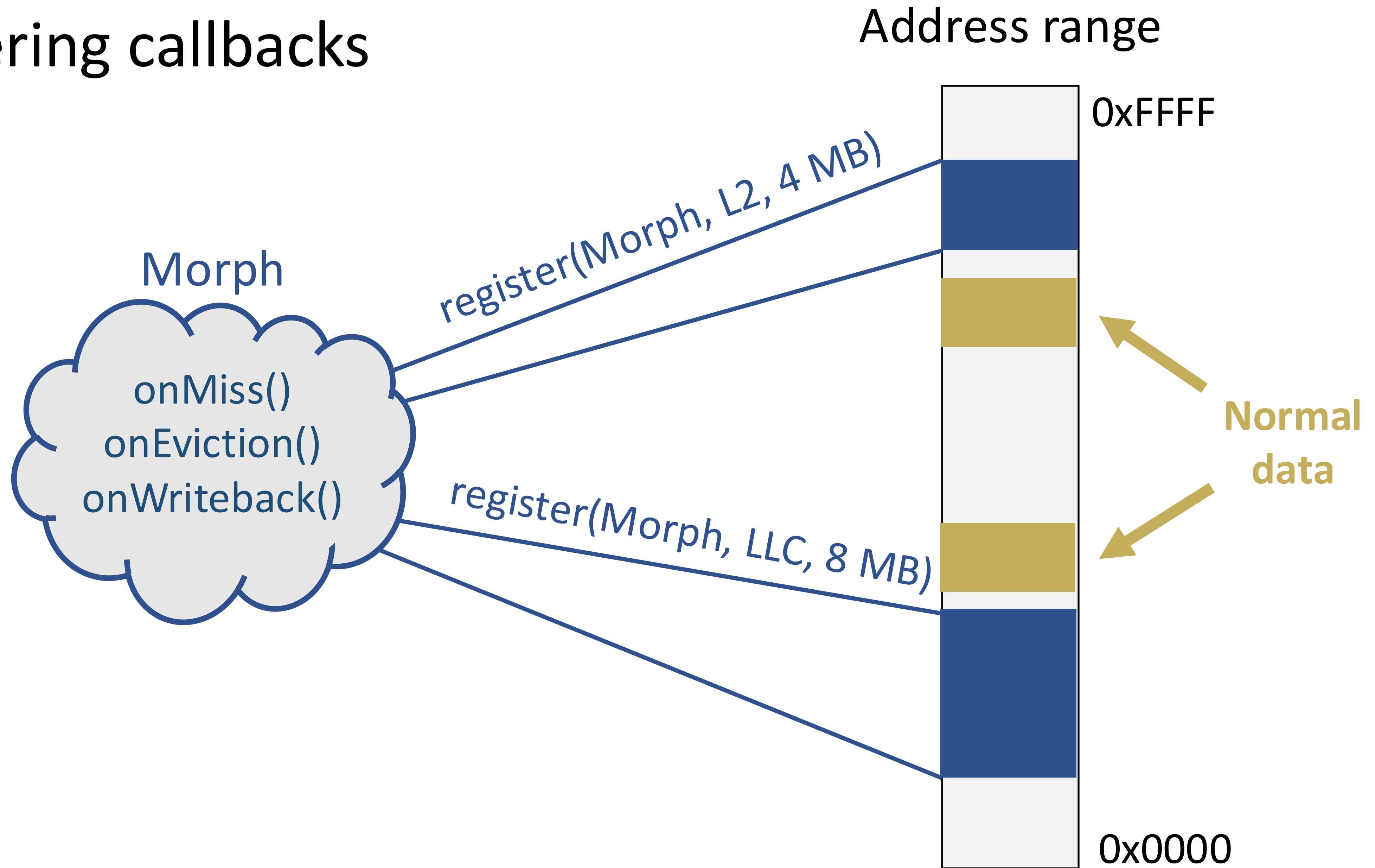
- Programming interface
- Architecture

Evaluation

täkō programming interface

Callback	Description
onMiss(void* addr)	Generate & fill in cache line at addr.
onEviction(void* addr)	Handle eviction of clean data.
onWriteback(void* addr)	Handle eviction of dirty data.

Registering callbacks



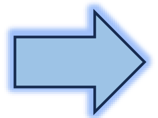
täkō: Roadmap

Introduction

Example use case

täkō design

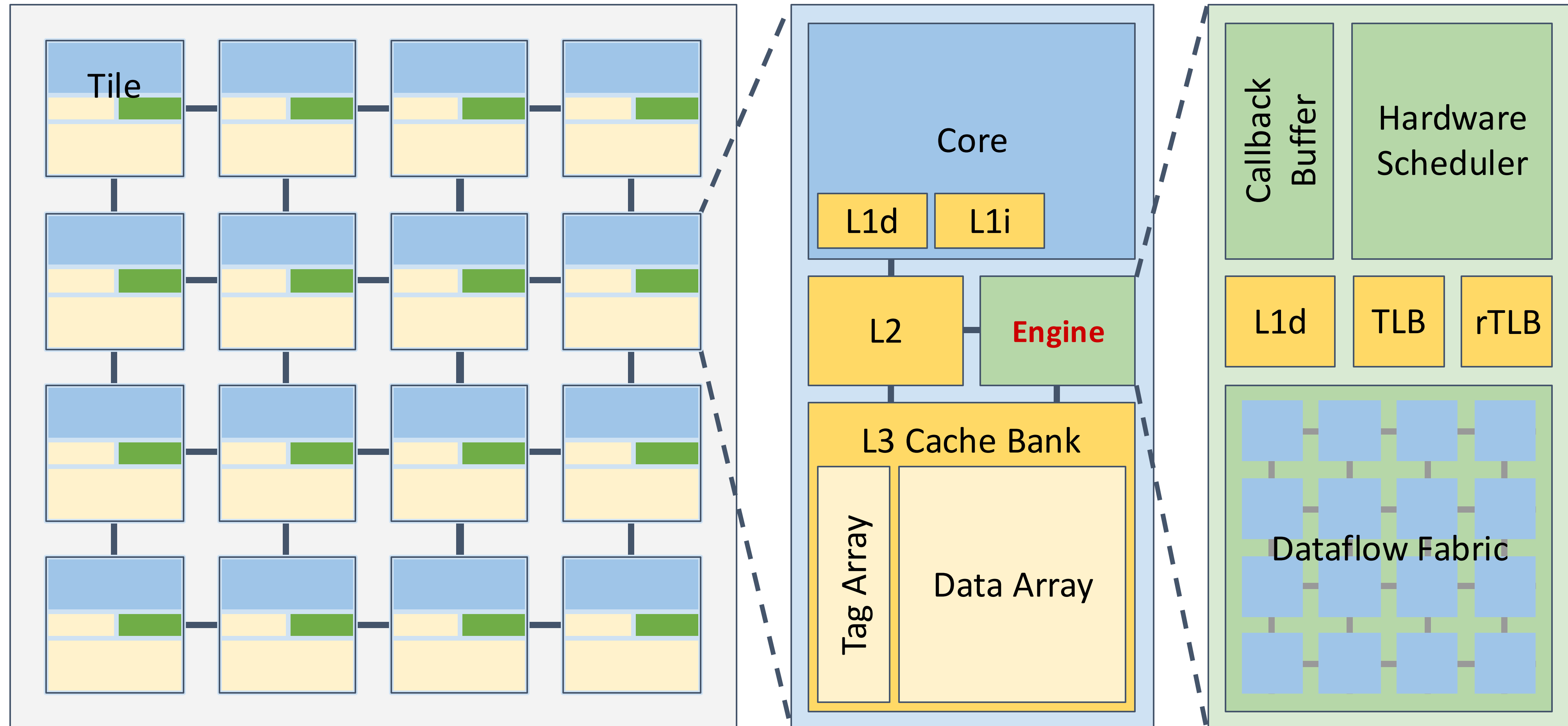
- Programming interface



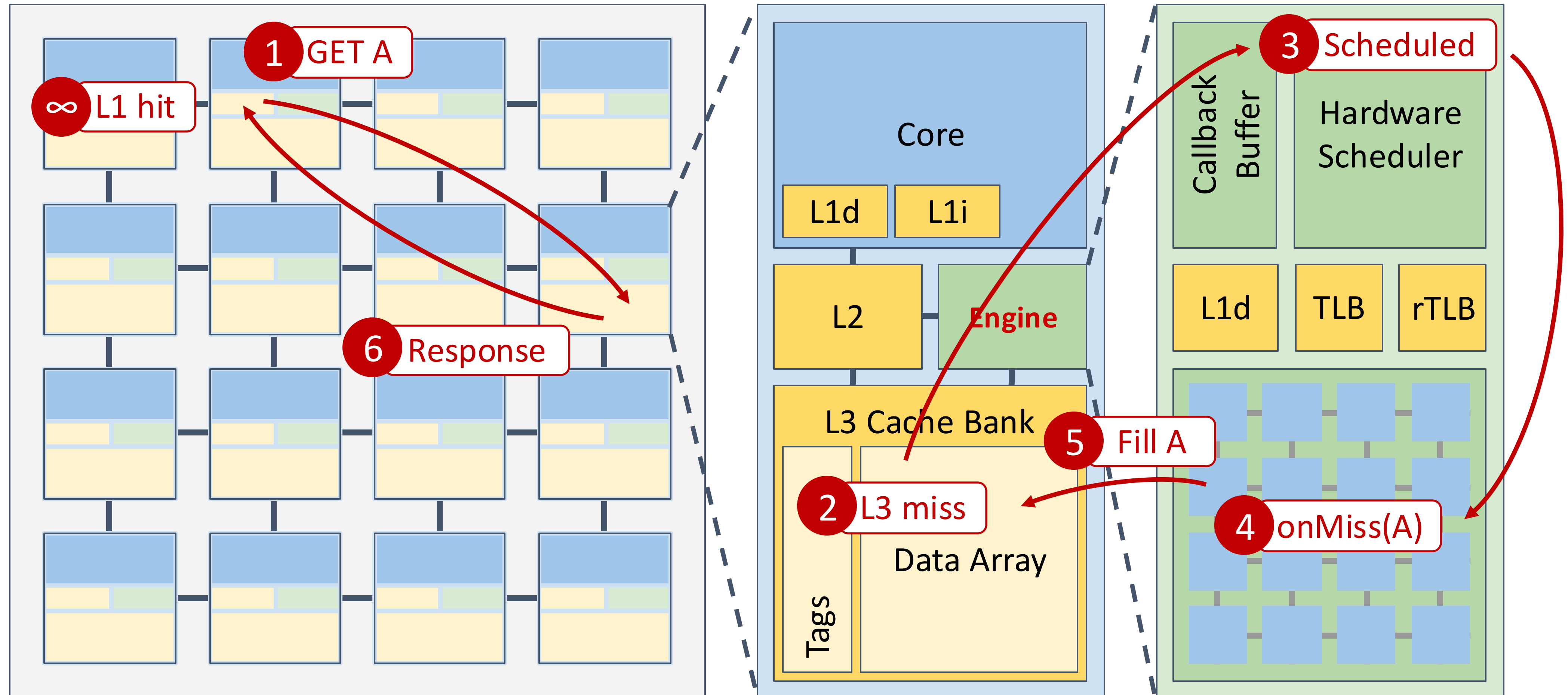
- Architecture

Evaluation

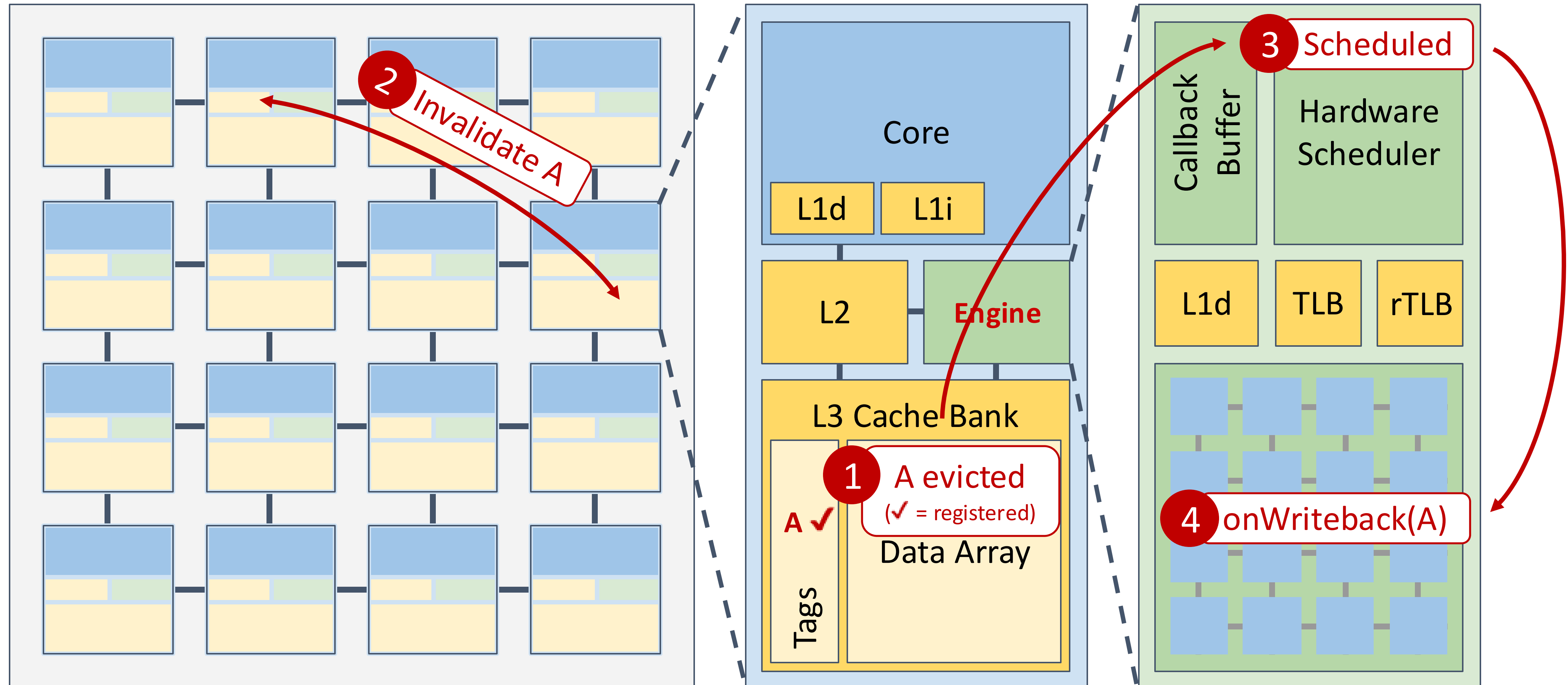
täkō architecture



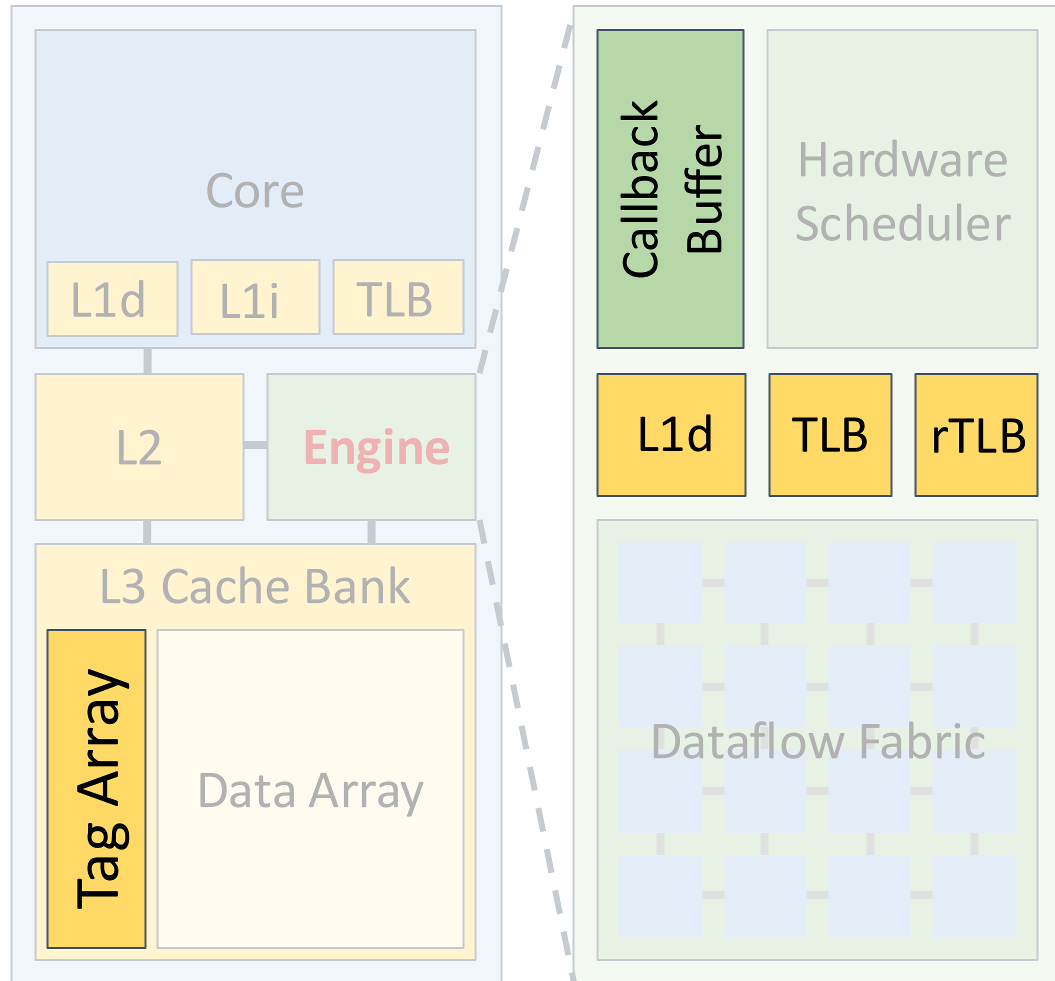
onMiss callback



onWriteback callback



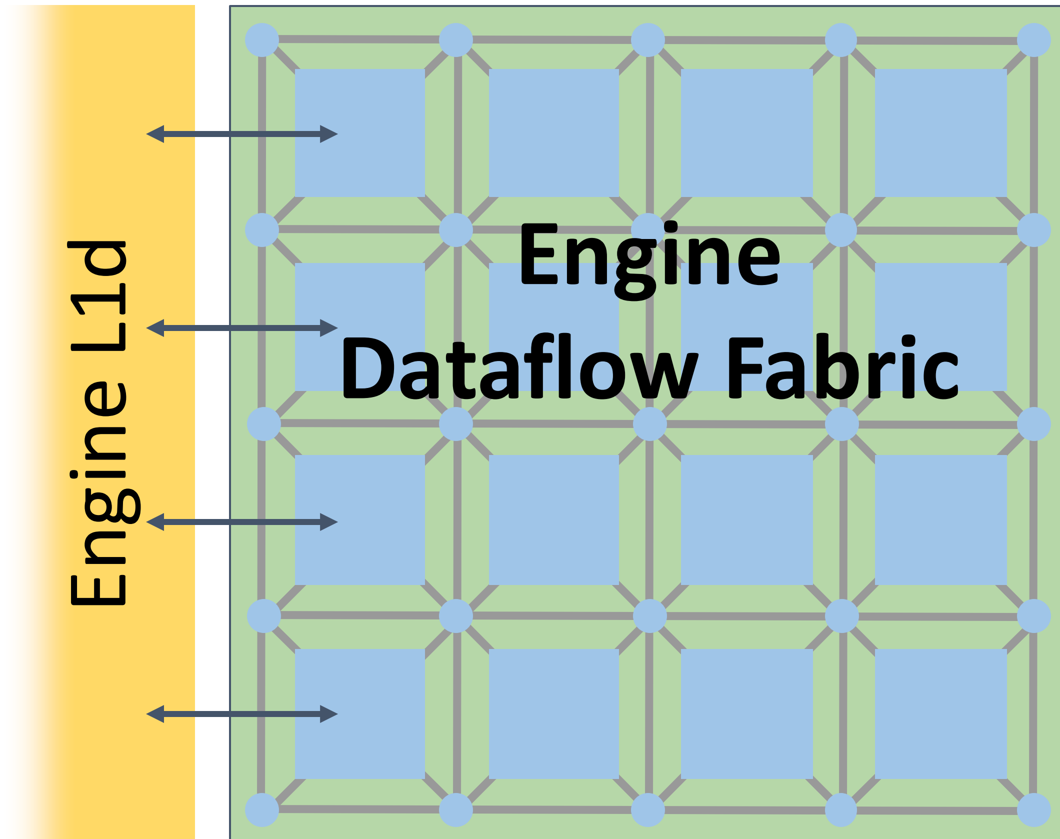
Additional cache and engine state



- 2 bits per TLB entry
- 1 bit per tag
- Callback buffer
- Engine L1d
- Engine TLB
- Engine reverse TLB

< 3% overhead vs. L3 cache

Engine microarchitecture



- Callbacks need memory-level parallelism
- Callbacks are typically short
- Callbacks are executed repeatedly
- Multiple callbacks execute in parallel

täkō: Roadmap

Introduction

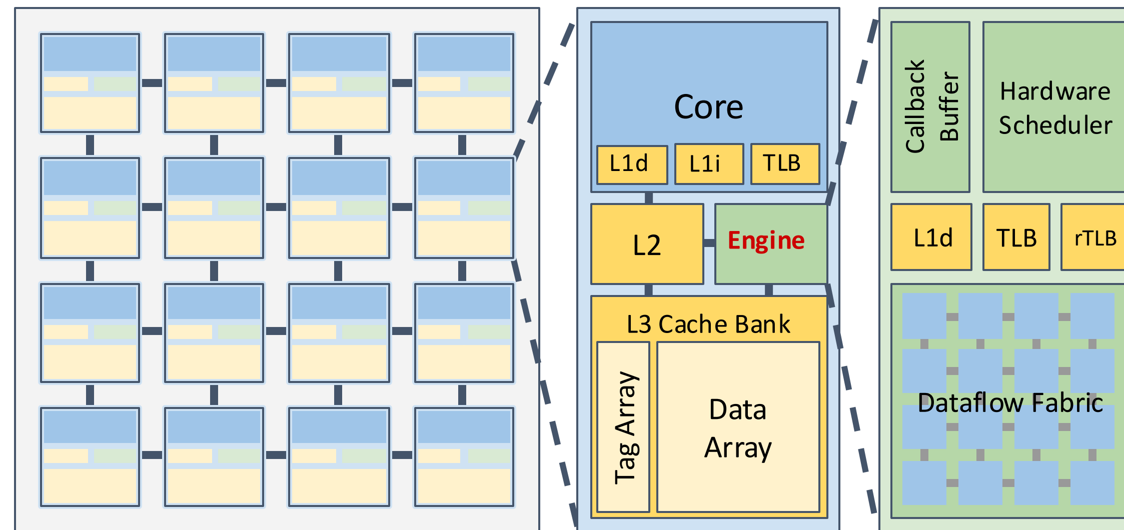
Example use case

täkō design

Evaluation

Evaluation methodology

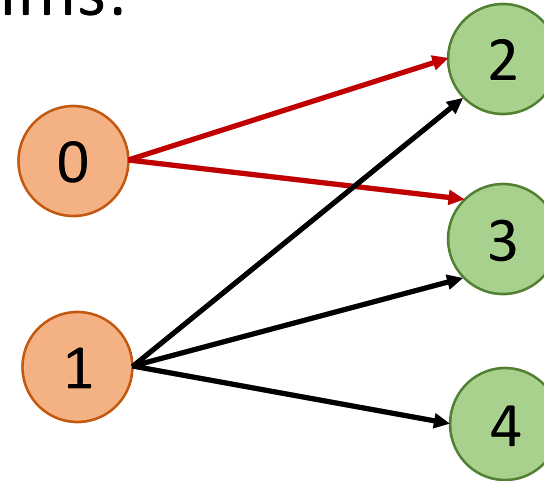
- Cycle-level simulation of 16-tile CMP
- Tile contains: OoO main core and täkō engine
- täkō dataflow fabric: 5x5 array, 1-cycle PE latency
- Ideal fabric: infinite PEs, 0-cycle PE latency



Commutative scatter-updates in graphs

Scatter-updates are common in graph algorithms:

```
for src in vertices:
    for dst in outNeighbors(src):
        vertex(dst) += vertex(src)
```



But current cache hierarchies are bad at scatter-updates

- Conflict between *pull* vs. *push* semantics
- ➔ Excess bandwidth & synchronization

[Mukkara, MICRO'19]

Commutative scatter-updates in graphs

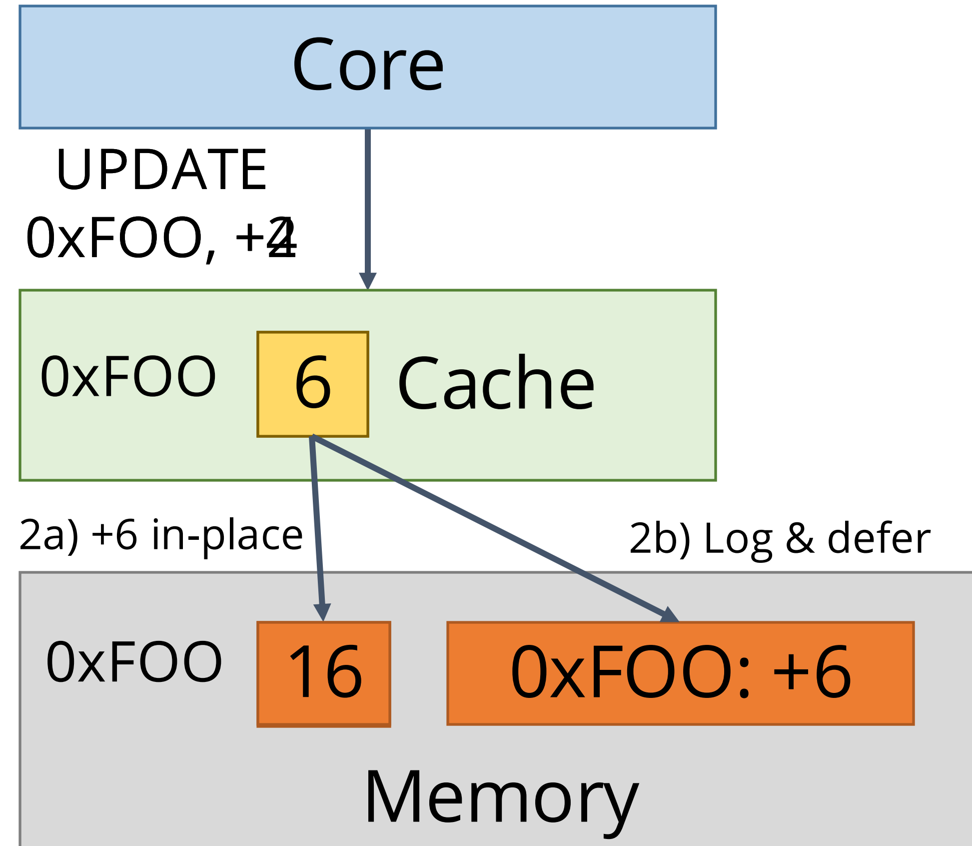
PHI changes the memory hierarchy to have **push-based semantics**

1. Updates are buffered in-cache without accessing memory

2. Updates applied on eviction:
   - 2a) Immediately in-place, or
   - 2b) Logged & deferred
     [Beamer, IPDPS'17]

Updates moved off critical path

Bandwidth optimized by 2a) vs 2b)



[Mukkara, MICRO'19]

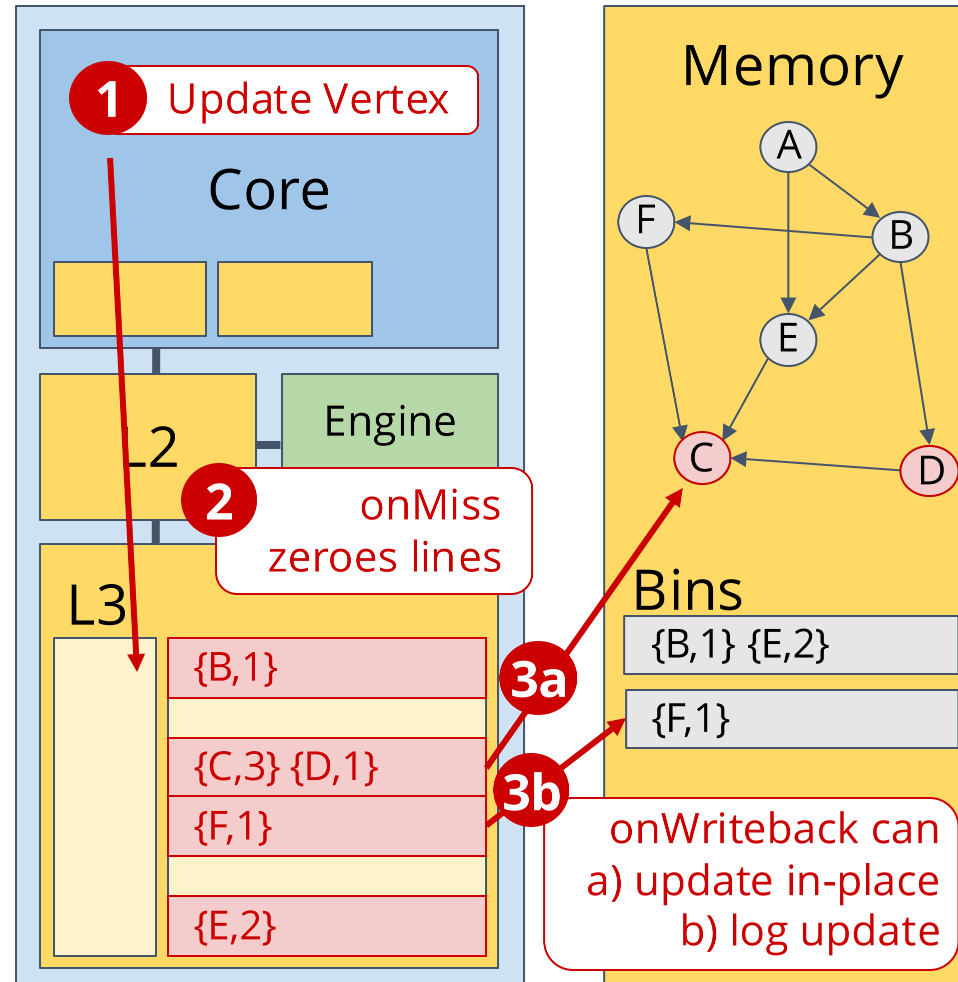
Example: PHI on täkō

Morph registered at L3 cache

Vertex updates buffered in phantom data

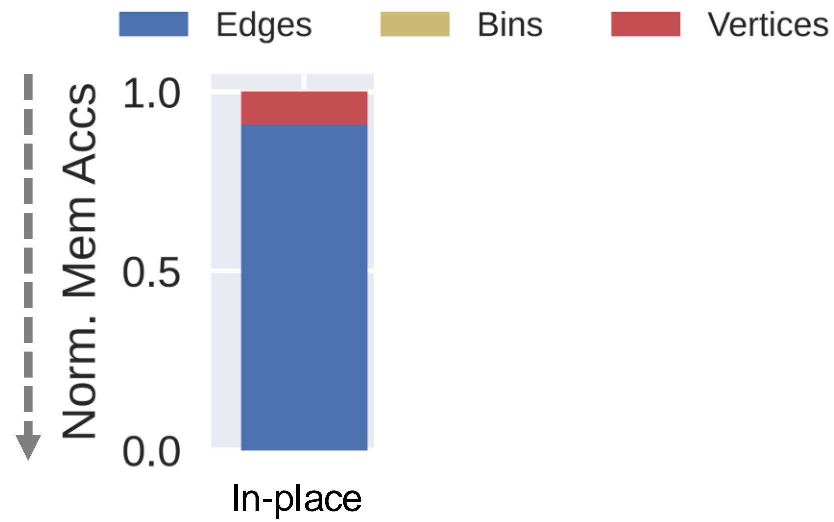
onWriteback selectively applies updates in-place or logs updates

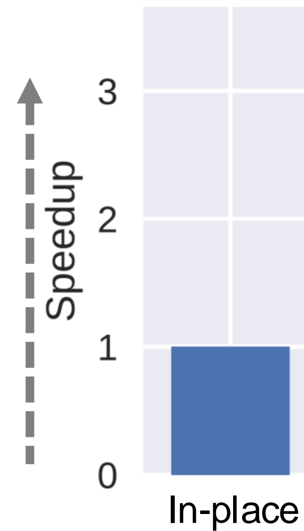
Impossible in software without täkō's callbacks!

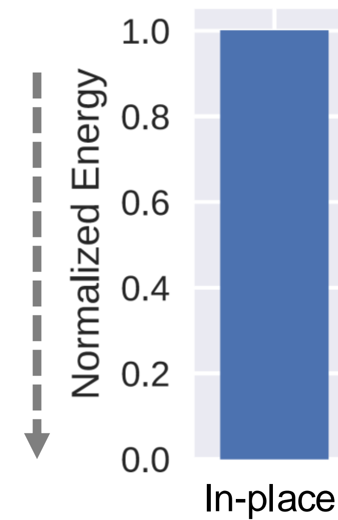


täkō's benefits implementing PHI

PageRank, 16 threads, graph: 8M vertices & 80M edges





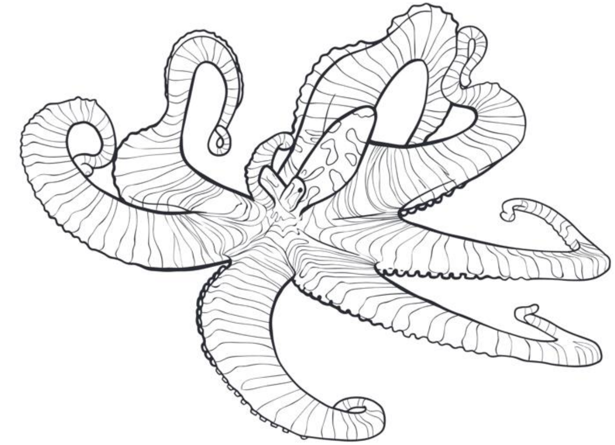


Software update batching (always log)

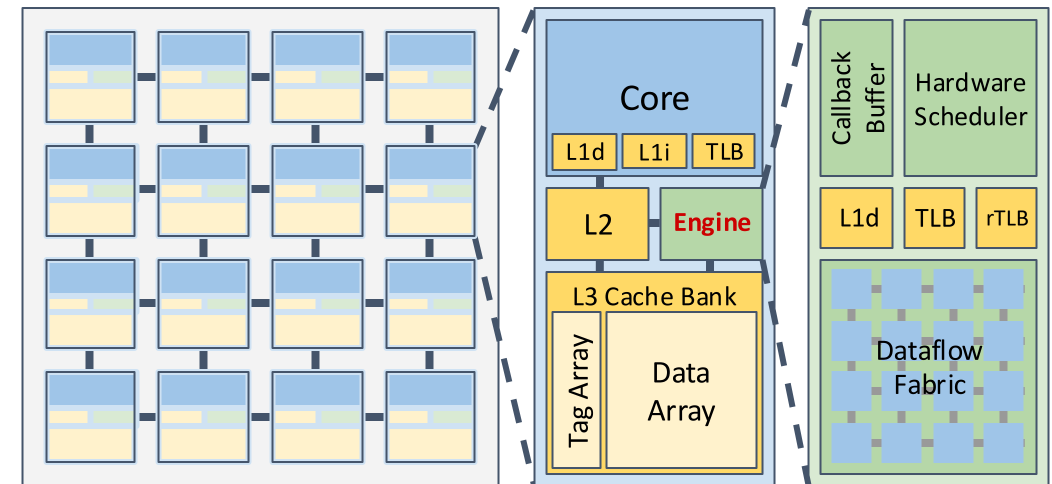
täkō with an ideal engine

[Beamer, IPDPS'17]

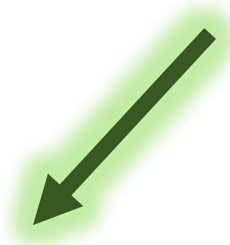
täkō: programmable specialized hierarchy



- The memory interface limits software capabilities
- Specialized hierarchies are beneficial but impractical
- täkō provides a general-purpose architecture to enable software optimization of the cache hierarchy

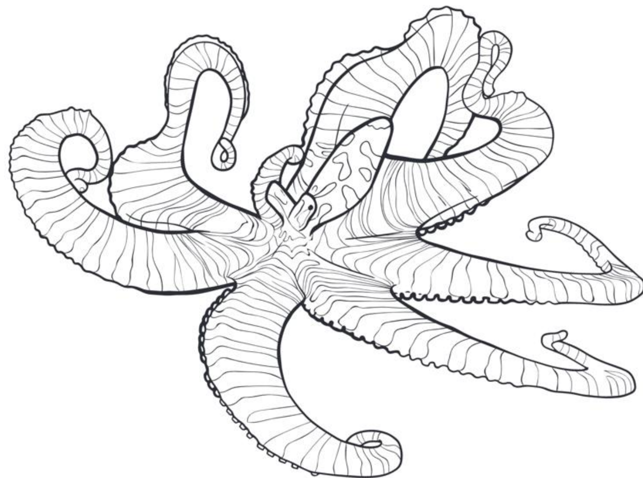


Overview



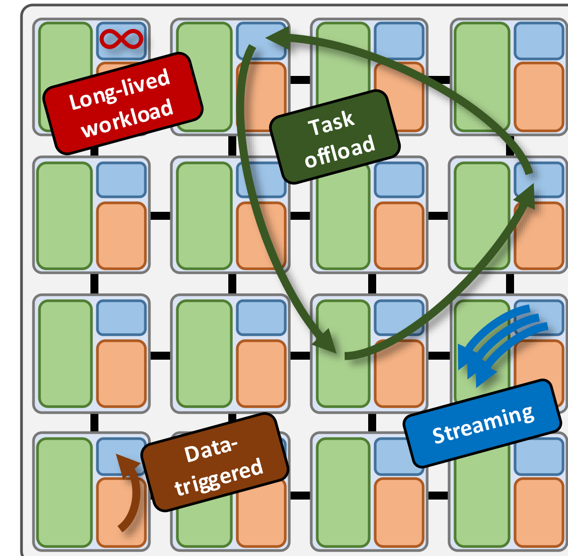
täkō

ISCA 2022

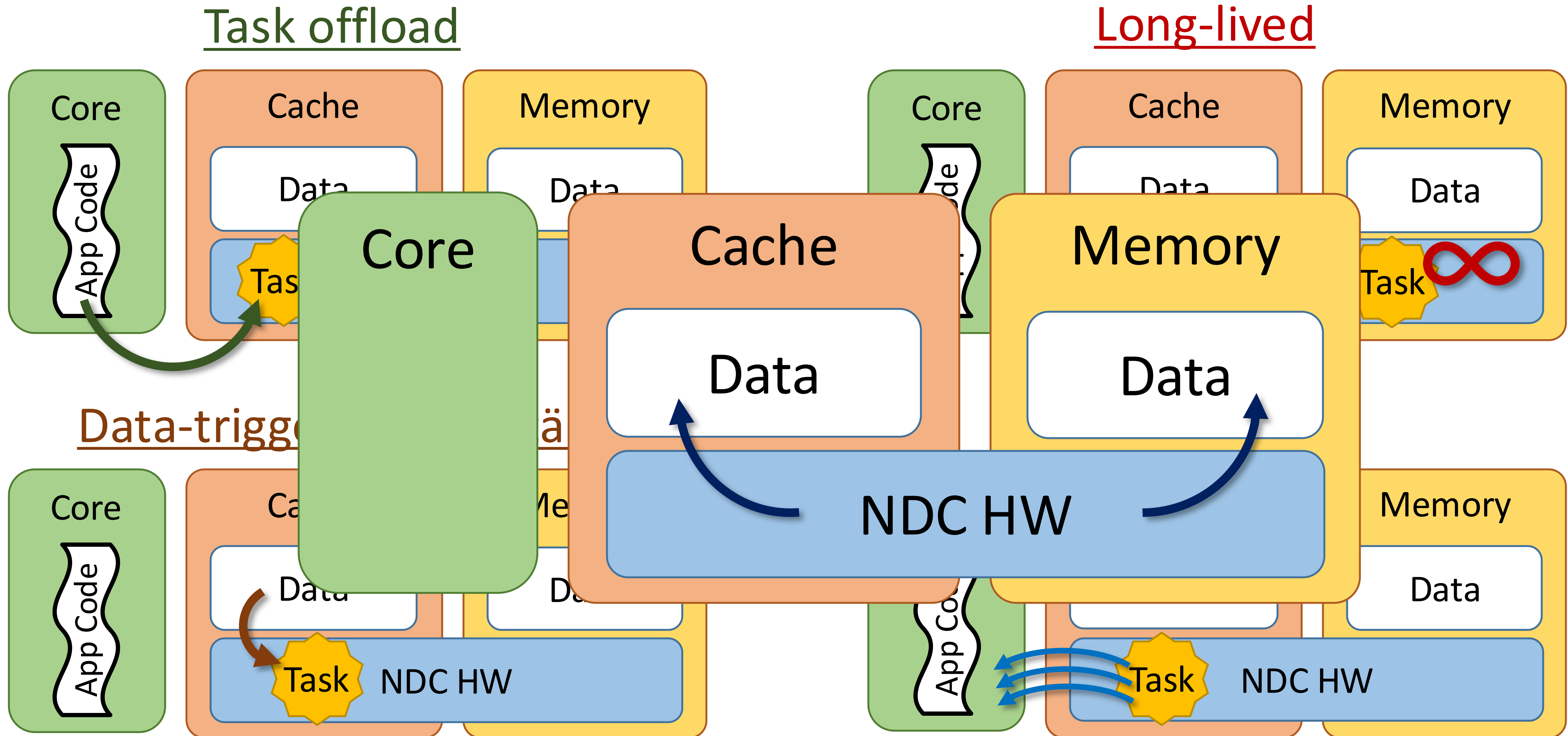


Leviathan

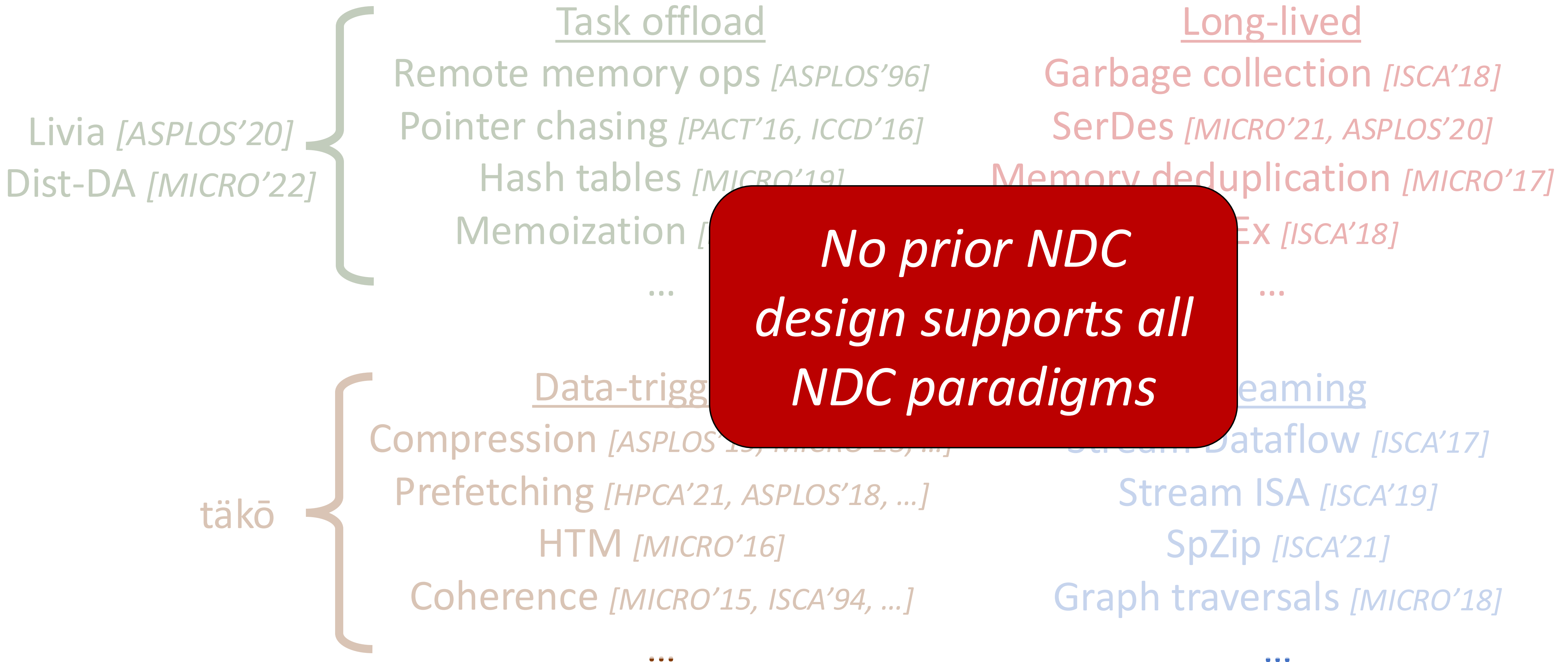
MICRO 2024



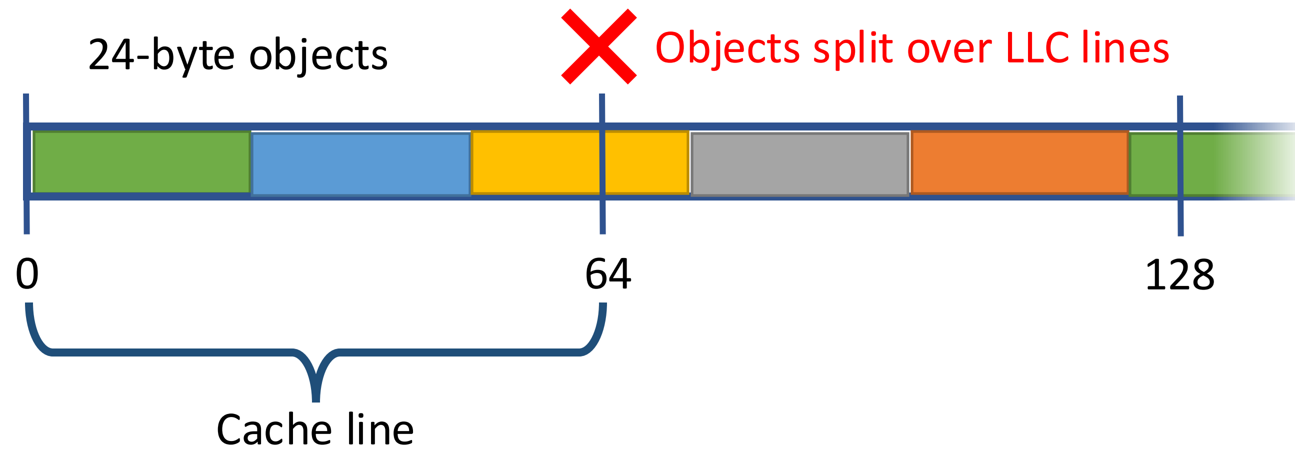
Near-data computing (NDC) paradigms

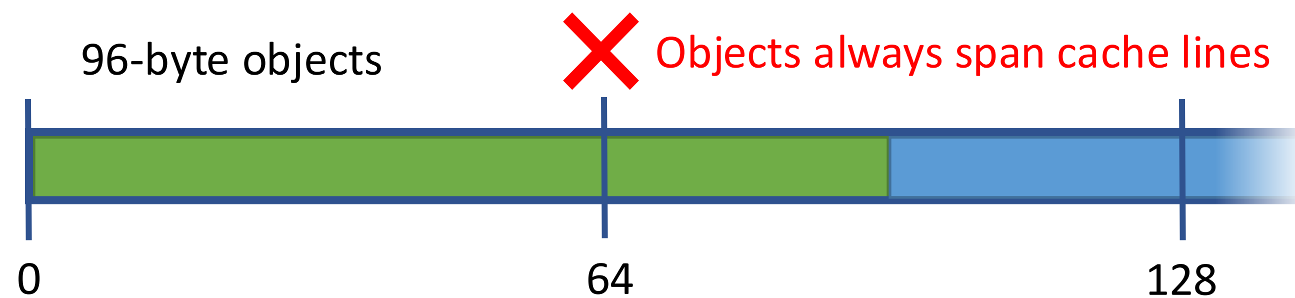


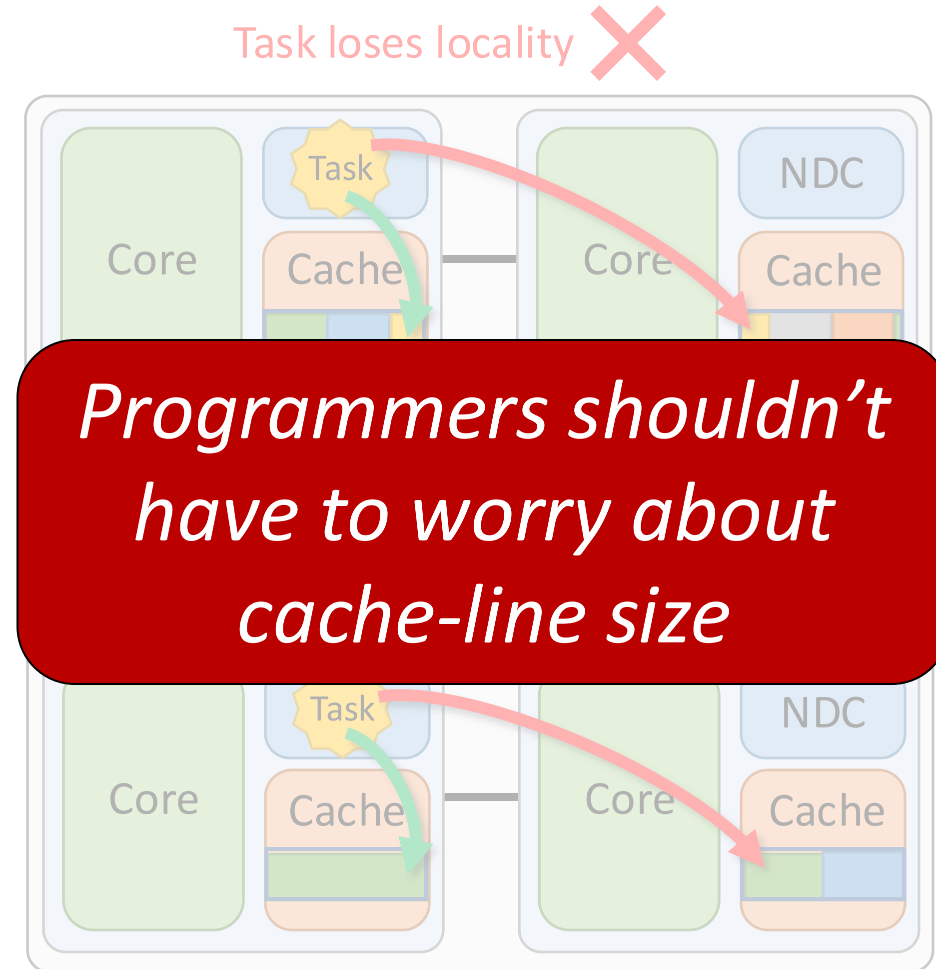
Prior NDC designs mostly target a single paradigm



Prior NDC suffers from ***poor hardware abstraction***

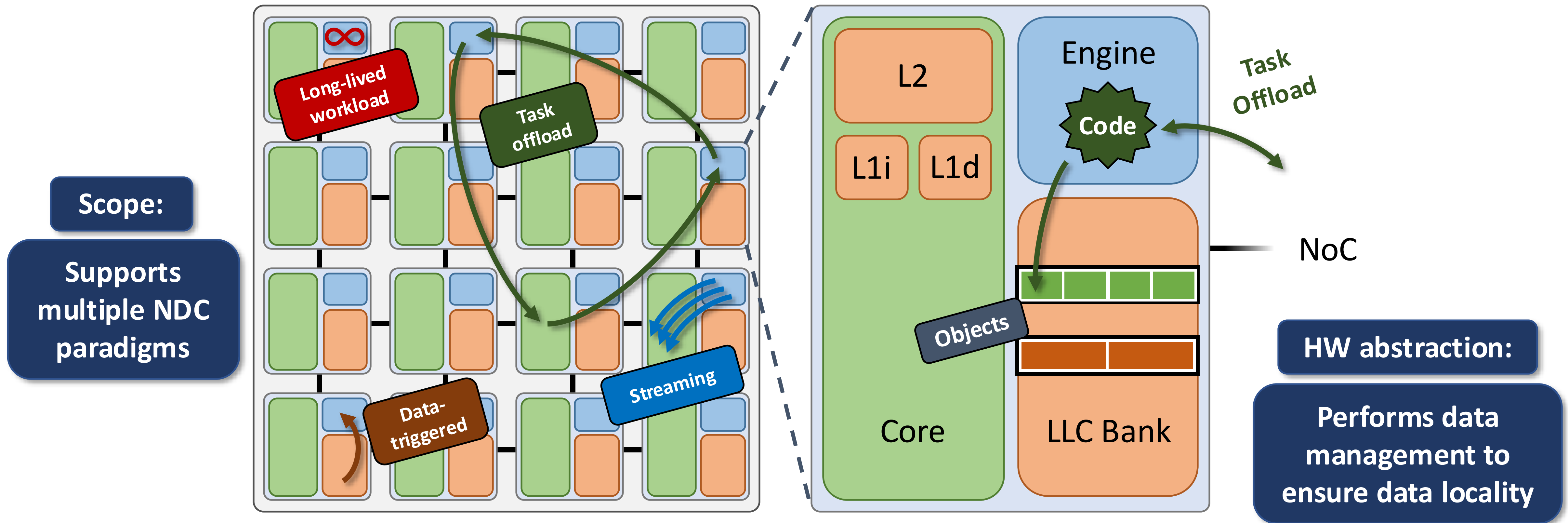






Programmers shouldn't have to worry about cache-line size

Leviathan: a unified NDC system with data management



Leviathan: Roadmap

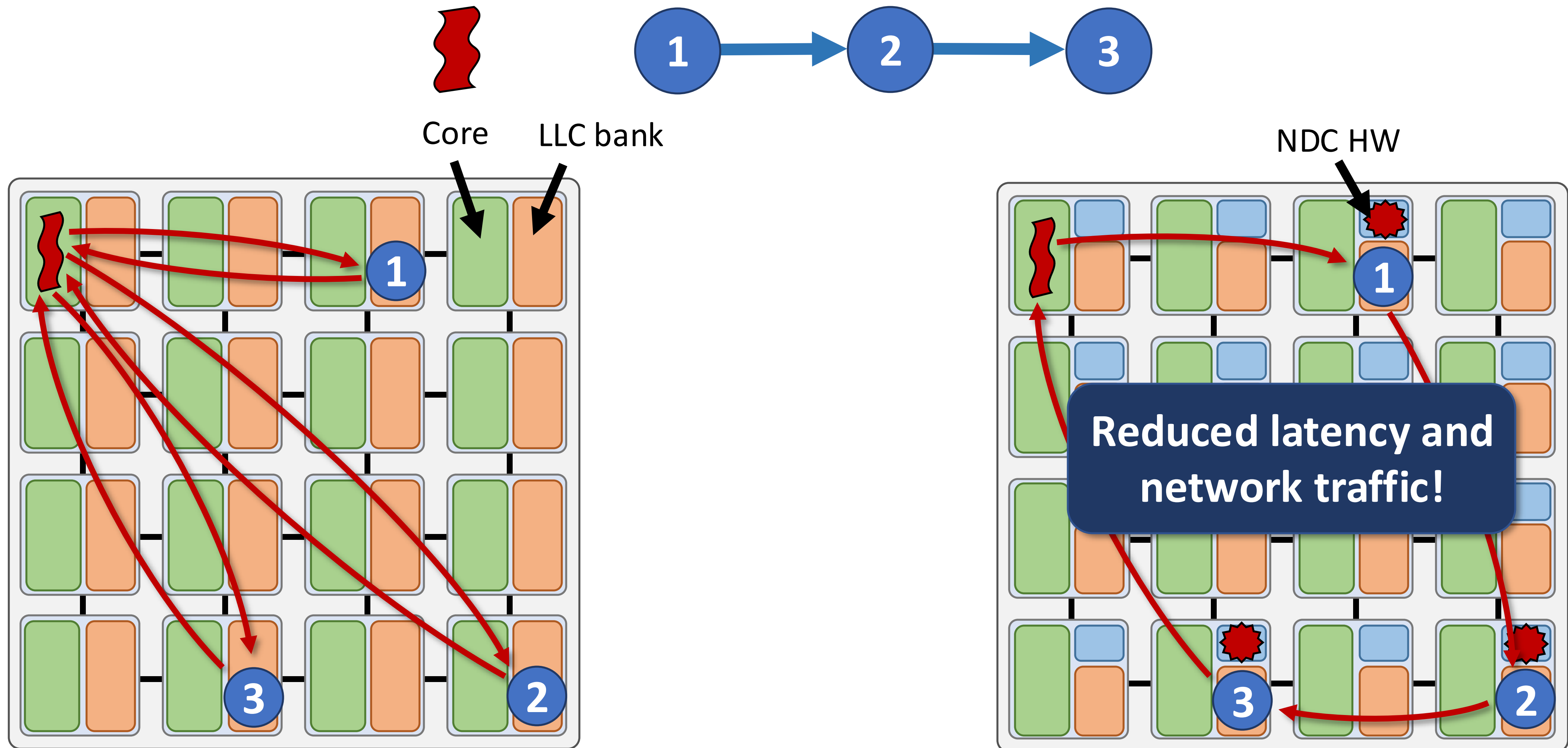
Motivation

Example NDC application

Design

Evaluation

Example application: linked-list traversal



Traditional: Each lookup pulls data to the core

Task offload NDC: Each lookup executes near the data

Leviathan: Roadmap

Motivation

Example NDC application

Design

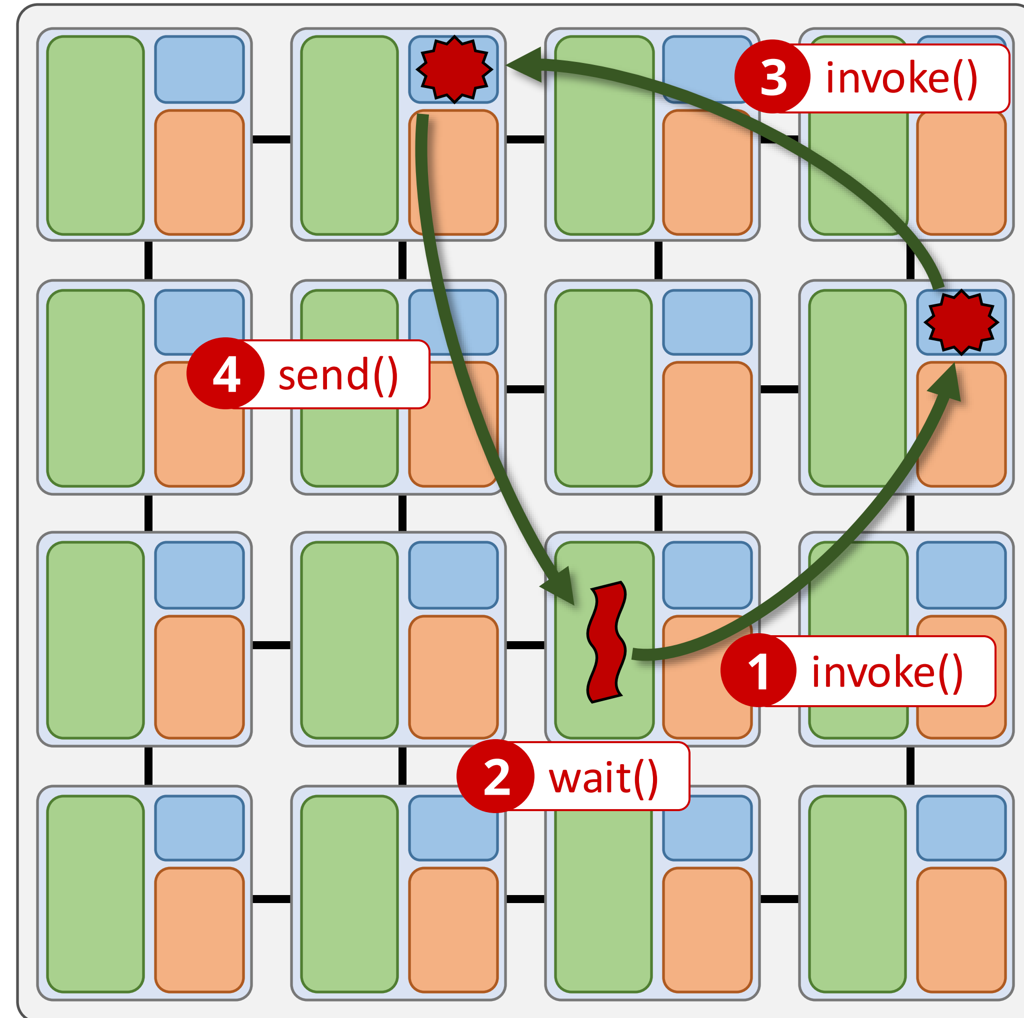
- Programming interface
- Architecture

Evaluation

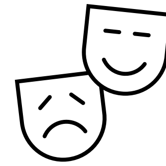
Programming interface: futures

Mechanism to asynchronously return data from NDC actions to the core

```
Future<T> fut = invoke (…)
T result = fut.wait()
```



Programming interface: actors

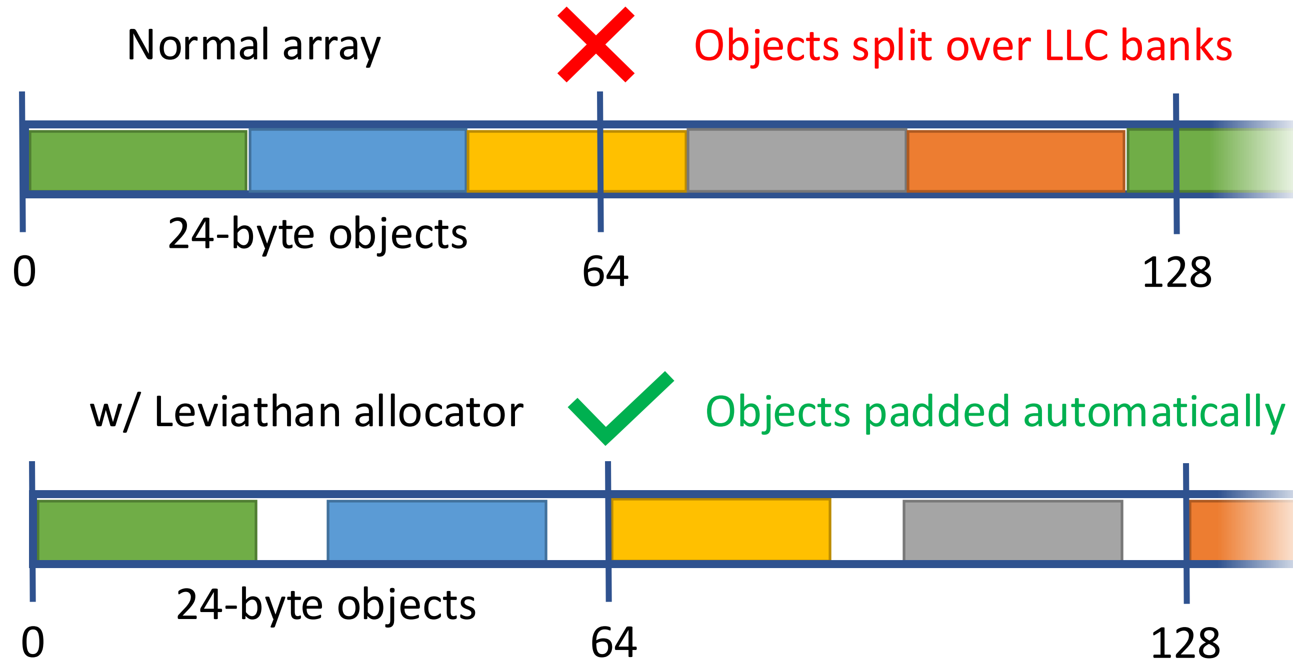


```
class Node:
    int key, value
    Node* next

    Node* Lookup(key):
        if (this->key == key):
            return value
        if (next == nullptr):
            return -1
        return invoke next->Lookup(key)

Future<int> result = invoke node->Lookup(key)
```

- `class Node:` ← **Actor = data + action(s)**
- `int key, value` / `Node* next` ← **Data**
- `Node* Lookup(key):` ← **Near-data action**
- `return value` ← **"return" sends the value to the future**
- `return invoke next->Lookup(key)` ← **Actions can trigger further actions**
- `Future<int> result = invoke node->Lookup(key)` ← **Example action triggered by core**

Programming interface: memory allocator

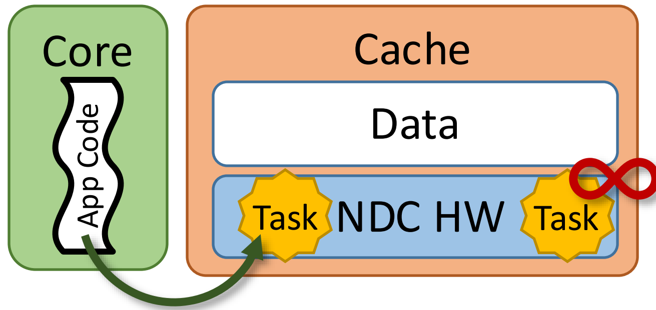
```
class Allocator<T>:
    T* allocate()
    void deallocate(T* object)
```



Leviathan only pads in cache!
Data is compacted in DRAM

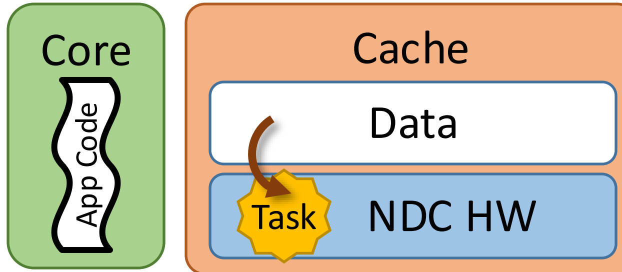
Programming interface: paradigms

Task offload & long-lived



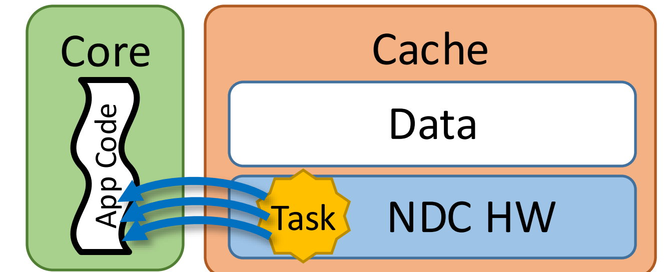

```
class Actor:          # actor
    int action1(…)    # action
    void action2(…)   # action

Actor* a =
    Allocator<Actor>::allocate()

Future<int> result =
    invoke a->action1(…)
```

Data-triggered




```
class Actor: # actor
    # actions
    Actor(State* s)
    ~Actor(State* s,
       bool isDirty)
```

Streaming




```
class Stream<T>: # actor
    # Producer interface
    void genStream() # action
    void push(T entry)

    # Consumer interface
    Future<T> next()
```

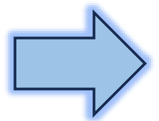
Leviathan: Roadmap

Motivation

Example NDC application

Design

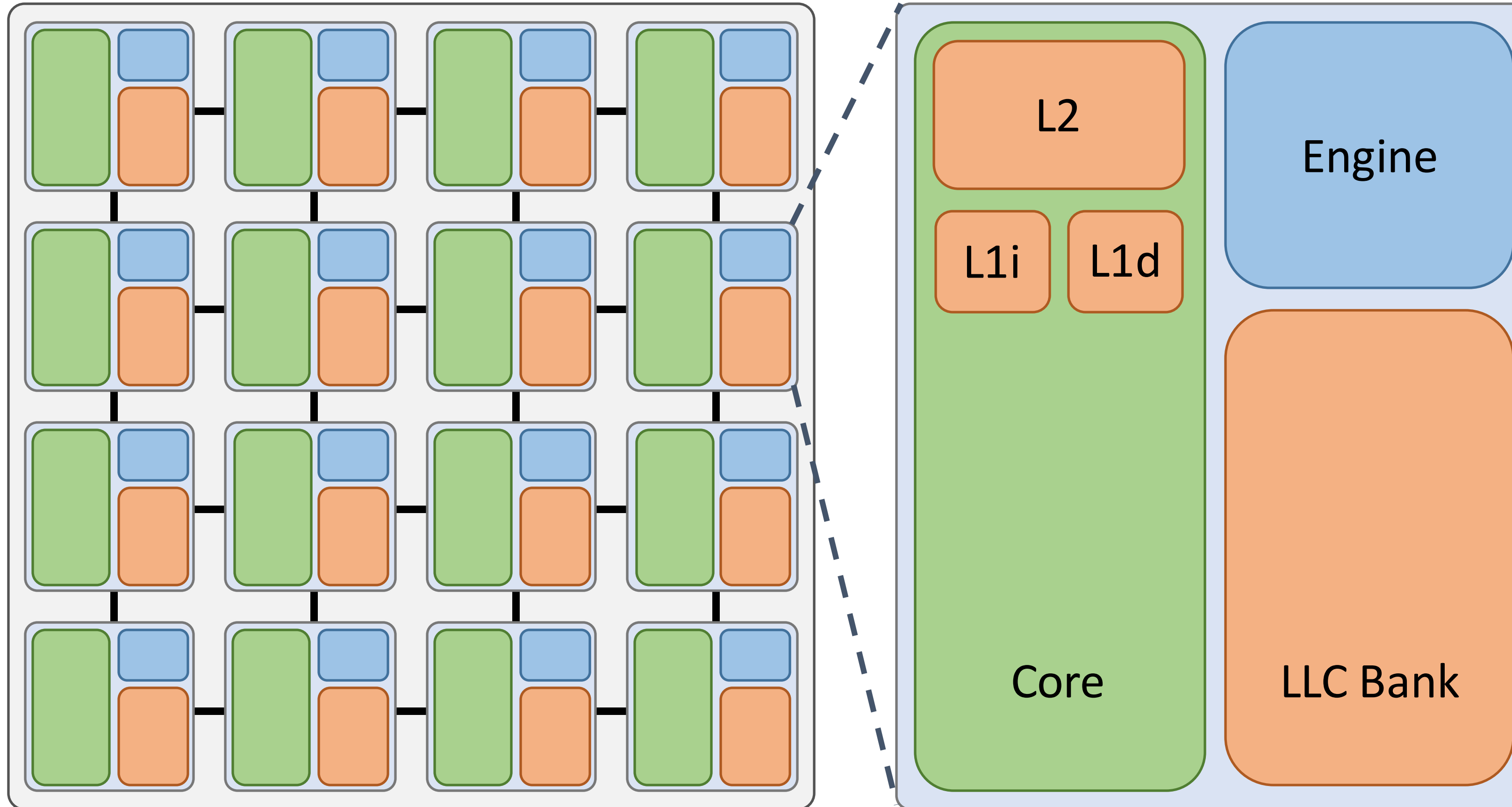
- Programming interface



- Architecture

Evaluation

Architecture: system

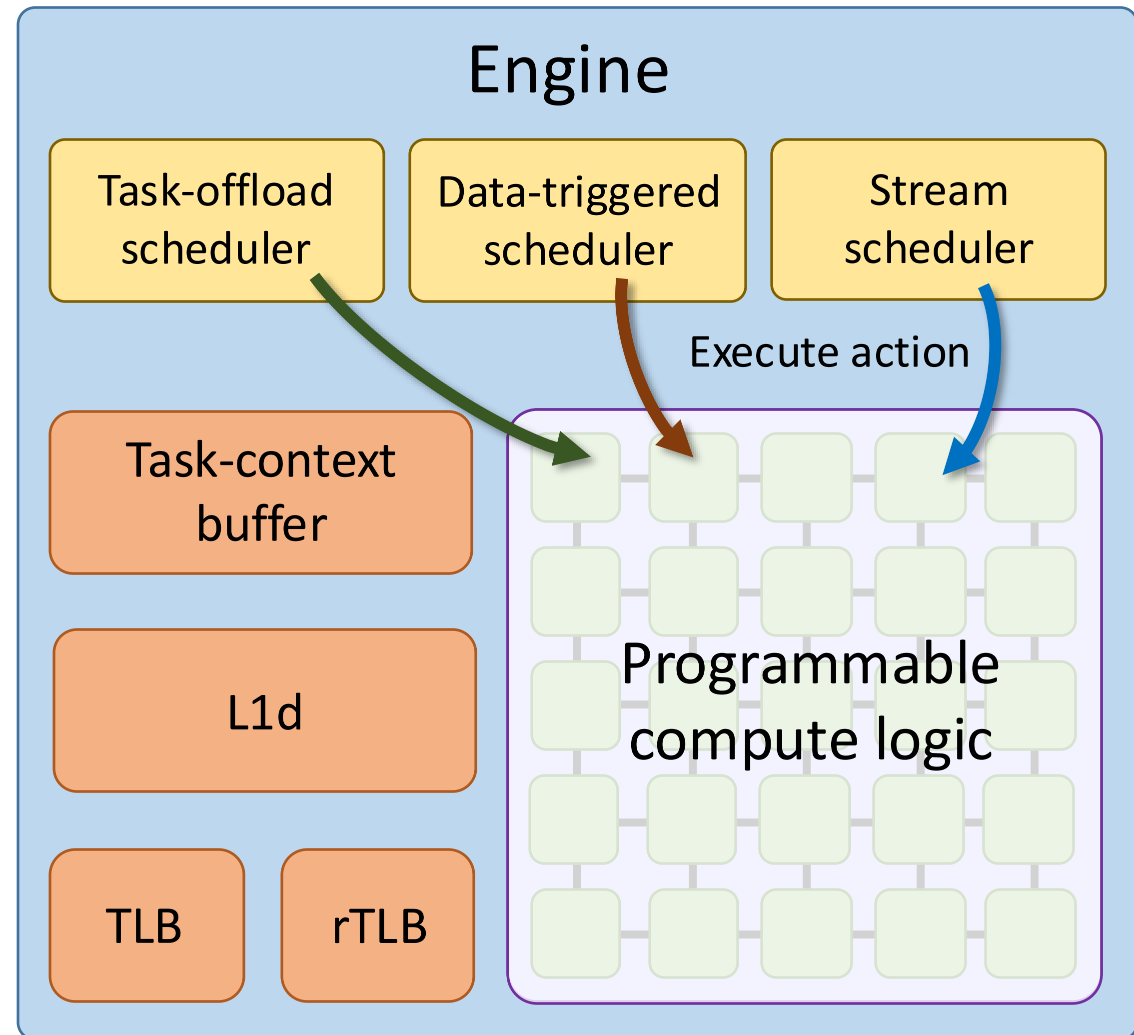


Architecture: engines

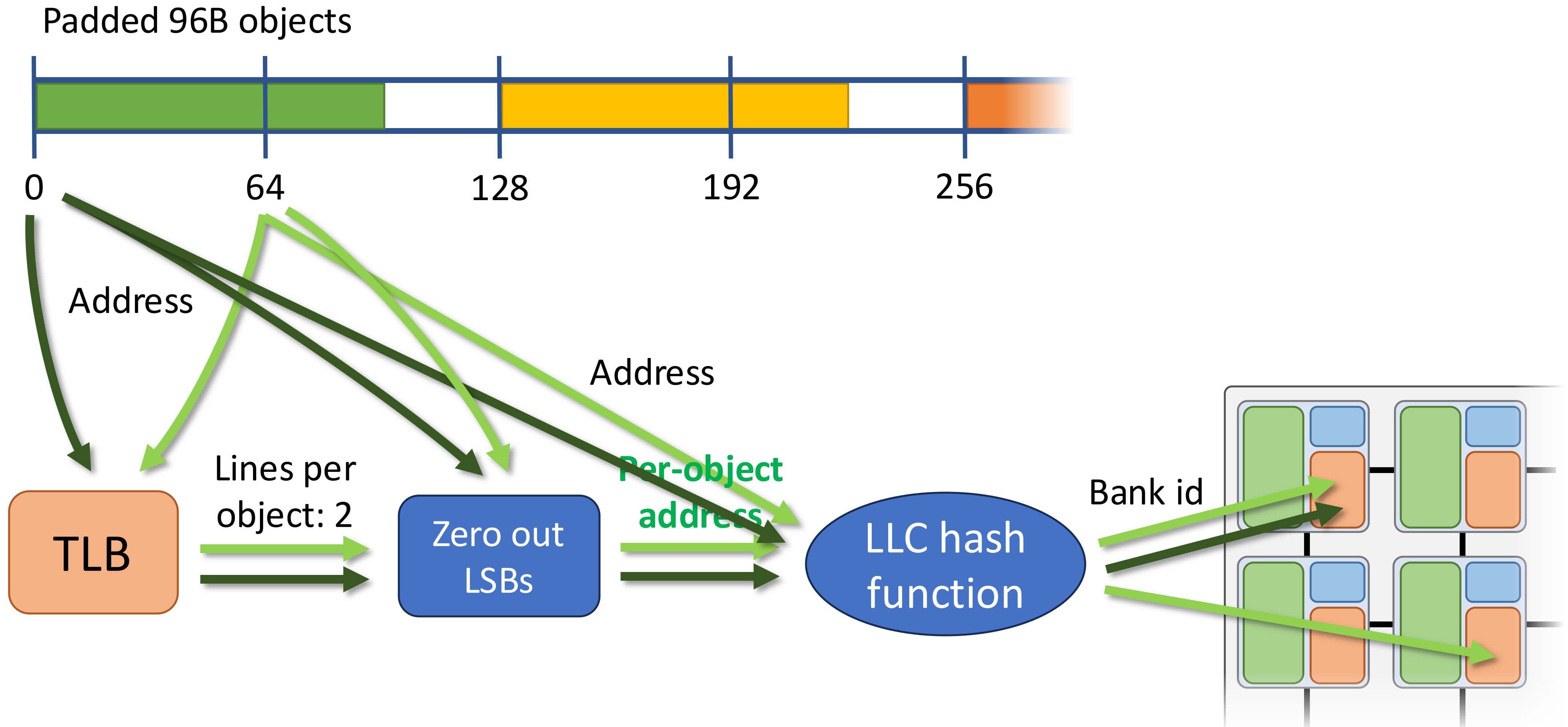
Task offload & long-lived:
- Check whether data is locally cached to execute action
- If data not cached, forward action to next level

Data-triggered:
- Maintain a buffer of currently-active actors
- Map actors to their actions

Stream:
- Track buffer size and head/tail pointers
- Stall producer when buffer is full



Architecture: large object mapping



Architecture: additional support

Compress padded data in main memory

Task offload & long-lived

- Invoke instruction
- Invoke backpressure buffer

Total HW overhead including engines: 6.4% vs LLC

Data-triggered

- 2 TLB registration bits
- 1 bit per cache tag

Streaming

- Push and pop instructions

Leviathan: Roadmap

Motivation

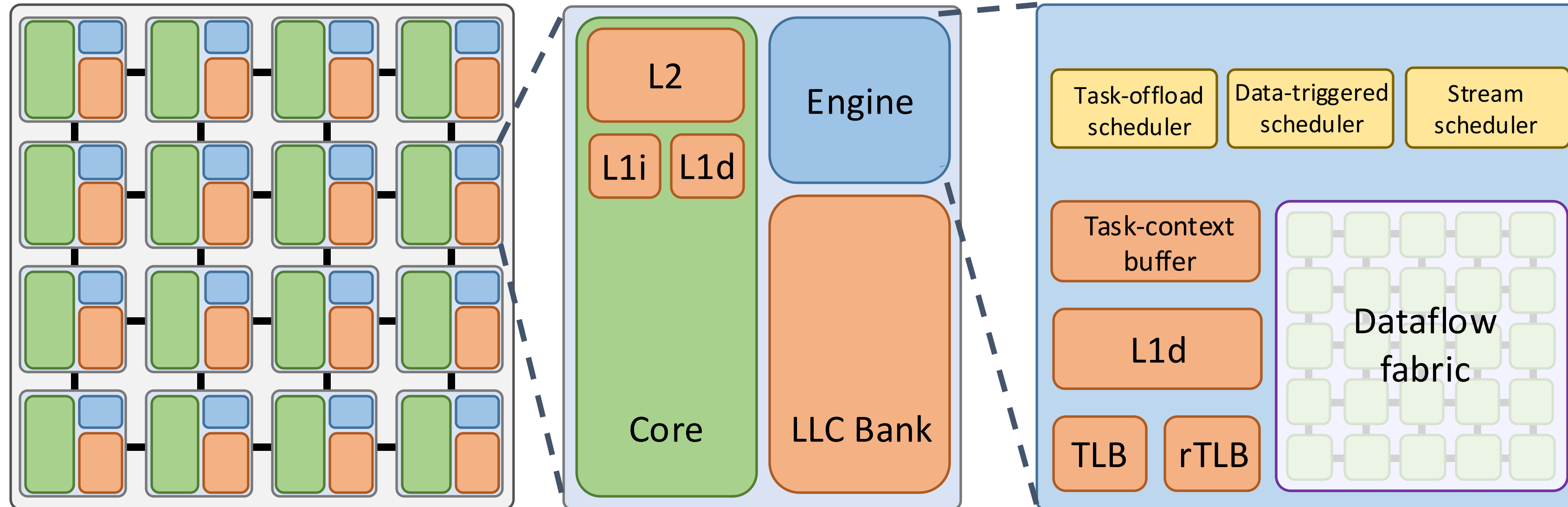
Example NDC application

Design

Evaluation

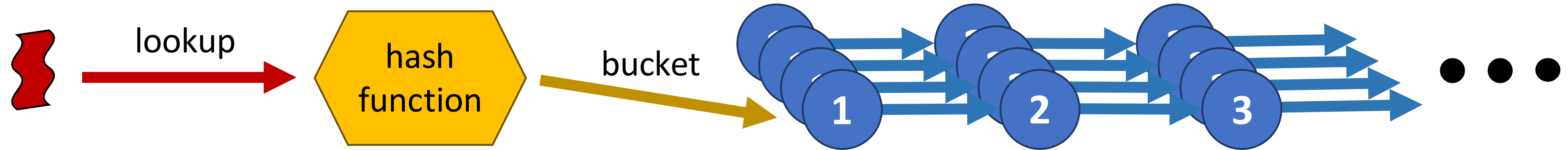
Evaluation methodology

- Cycle-level simulation of 16-tile CMP
- Tile contents: OoO main core, LLC bank, and Leviathan engine
- Engine compute: dataflow fabric, 5x5 array, 1-cycle PE latency
- Ideal engine: dataflow fabric, infinite PEs, 0-cycle PE latency



Case Study:
Hash-table Lookups

Hash-table lookups w/ task offload NDC

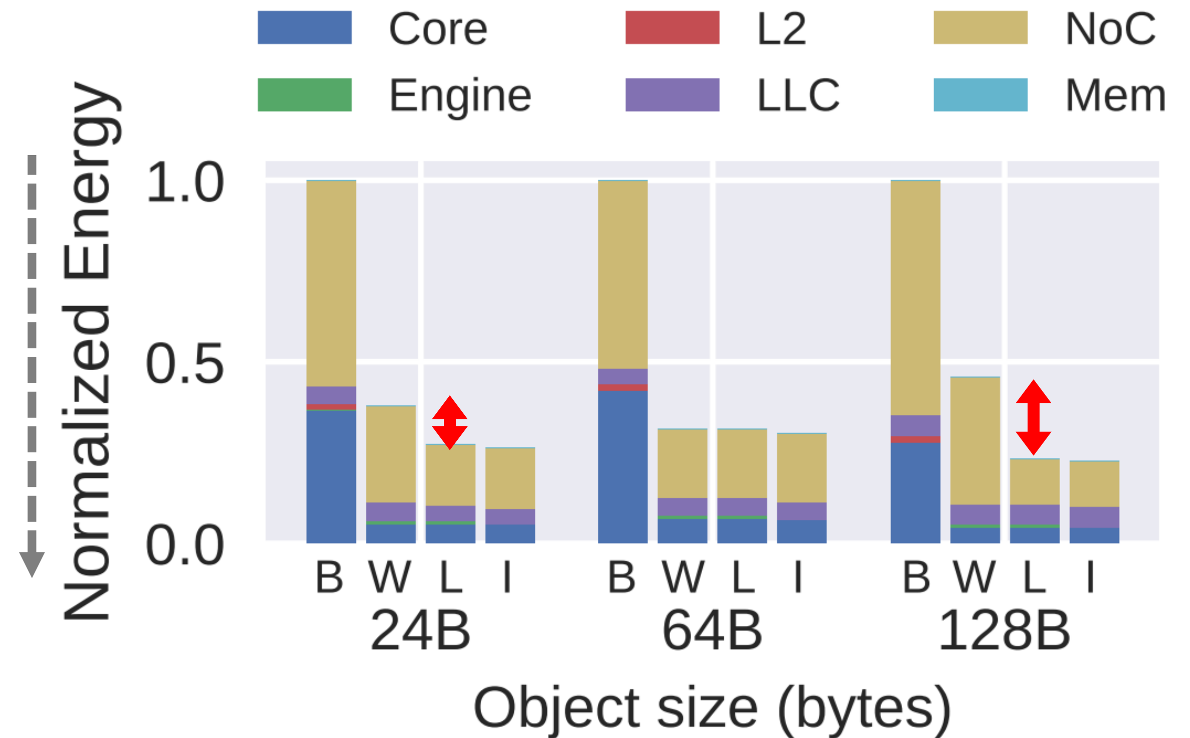



```
class Node:
    Node* next
    int key
    int values[N]
```

Data size doesn't matter

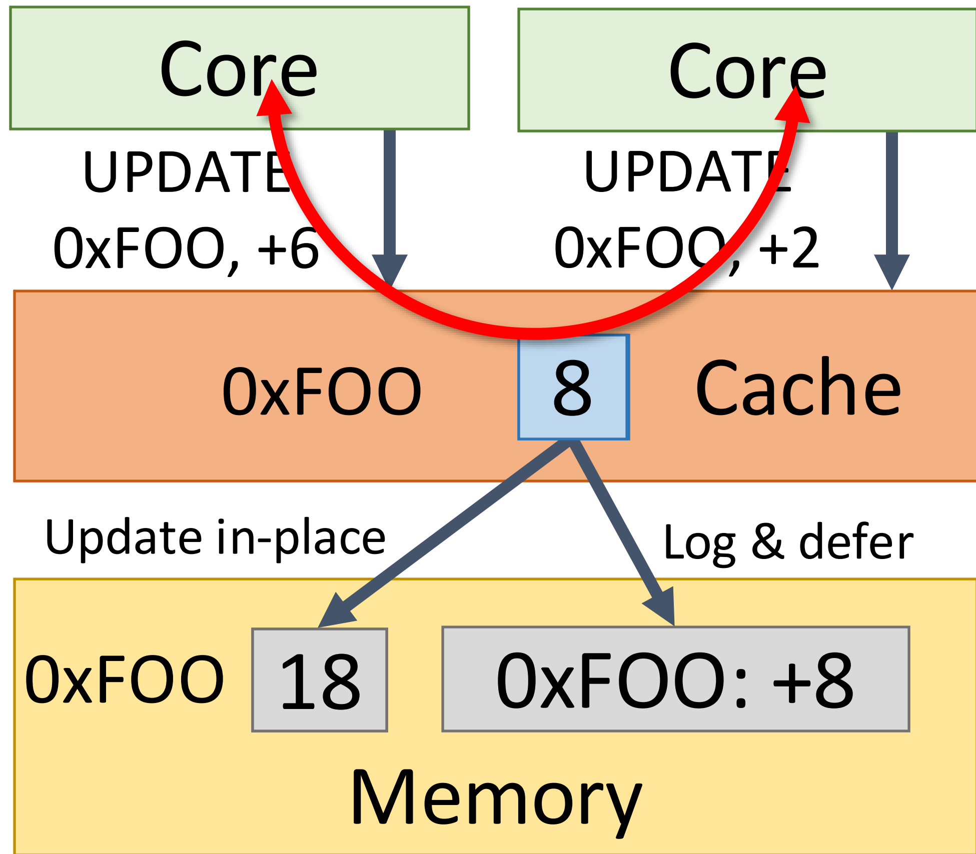
Leviathan's benefits with transparent data management

16 threads, 32-node buckets, 4 MB total, lists accessed with uniform distribution



Case Study:
Multi-paradigm Accelerator

PHI requires support for multiple NDC paradigms



Task offload:
Offload remote memory operation (RMO) to cache

NOT supported by täkō

Data-triggered:
Conditionally update or log on evictions

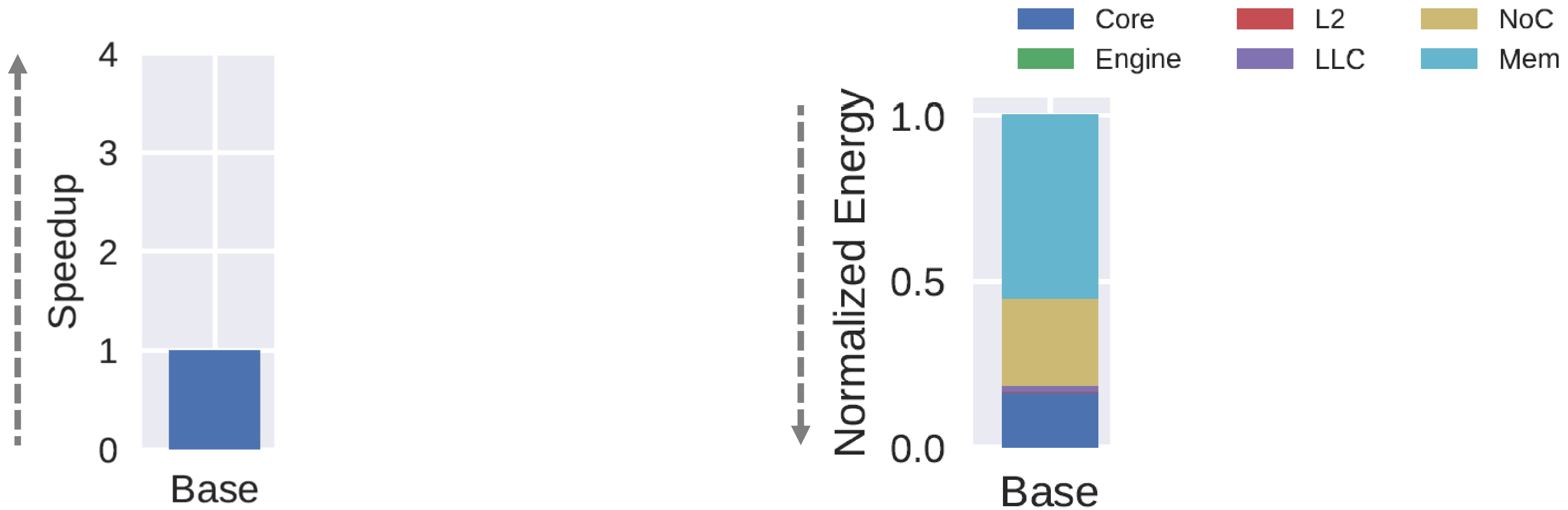
Supported by täkō

Leviathan is the only programmable system that supports both paradigms!

[Mukkara, MICRO'19]

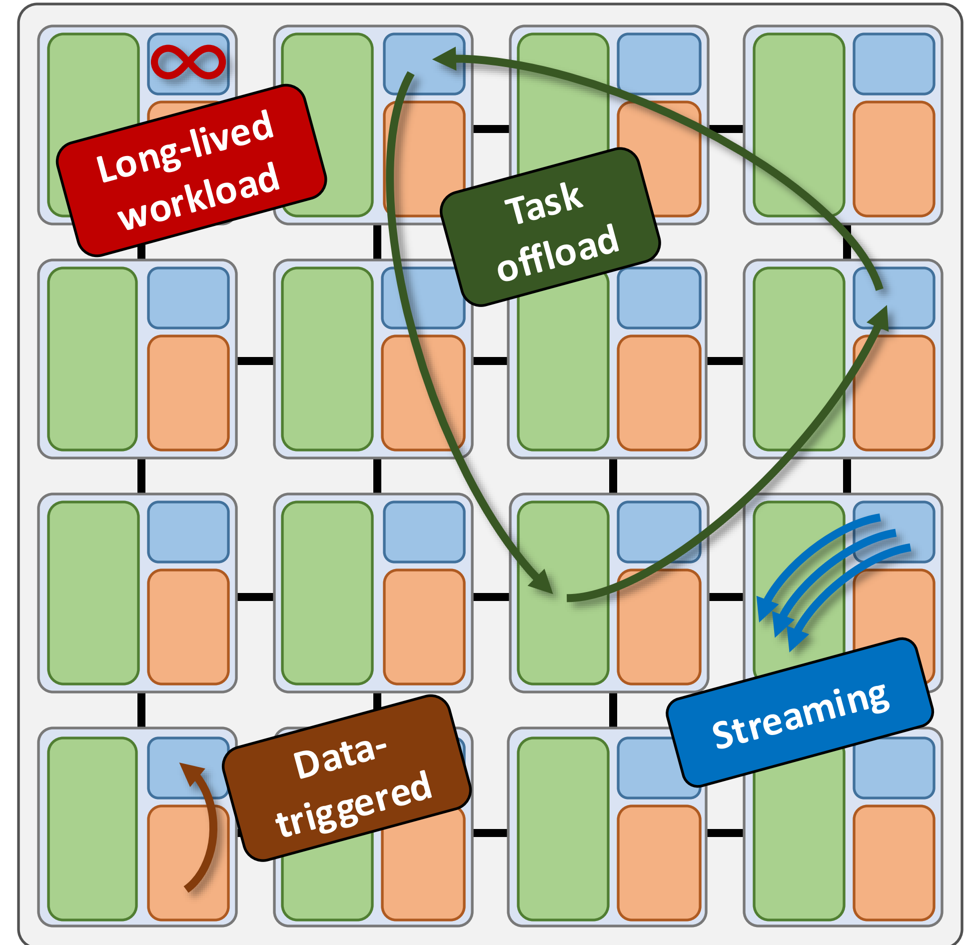
Leviathan’s benefits supporting multiple paradigms w/ PHI

PageRank, 16 threads, graph: 4M vertices & 40M edges



Leviathan: unified system for NDC

- Practical NDC systems need to support many types of NDC
- Data management is vital to provide locality to near-data actions
- Leviathan provides a single architecture and programming interface for near-data computing



Takeaways

- Near-data computing tackles the data-movement bottleneck
- Custom hardware and programming interfaces for each optimization is not scalable
- General-purpose architectures enable software to implement a wide range of NDC designs