täkō: A Polymorphic Cache Hierarchy for General-Purpose Optimization of Data Movement

Brian C. Schwedock
Carnegie Mellon University
bschwedo@andrew.cmu.edu

Piratach Yoovidhya
Carnegie Mellon University
piratacy@andrew.cmu.edu

Jennifer Seibert
Binghamton University
jseiber1@binghamton.edu

Nathan Beckmann
Carnegie Mellon University
beckmann@cs.cmu.edu

ABSTRACT
Current systems hide data movement from software behind the load-store interface. Software’s inability to observe and respond to data movement is the root cause of many inefficiencies, including the growing fraction of execution time and energy devoted to data movement itself. Recent specialized memory-hierarchy designs prove that large data-movement savings are possible. However, these designs require custom hardware, raising a large barrier to their practical adoption.

This paper argues that the hardware-software interface is the problem, and custom hardware is often unnecessary with an expanded interface. The täkō architecture lets software observe data movement and interpose when desired. Specifically, caches in täkō can trigger software callbacks in response to misses, evictions, and writebacks. Callbacks run on reconfigurable dataflow engines placed near caches. Five case studies show that this interface covers a wide range of data-movement features and optimizations. Microarchitecturally, täkō is similar to recent near-data computing designs, adding ≈5% area to a baseline multicore. täkō improves performance by 1.4×–4.2×, similar to prior custom hardware designs, and comes within 1.8% of an idealized implementation.

CCS CONCEPTS
• Computer systems organization → Processors and memory architectures.

KEYWORDS
cache hierarchy, data movement, data-centric computing

ACM Reference Format:

1 INTRODUCTION
Data movement dominates computer systems’ performance and energy efficiency, and it is only getting worse over time [30, 53, 55, 76]. Ideally, hardware and software would work together to optimize data movement. But mainstream instruction set architectures were designed at a time when data movement was inexpensive and do not emphasize it. In current systems, software reads and writes data, and hardware decides when and where to move it.

Lacking visibility and control over data movement, software cannot implement many attractive features or optimizations, and instead resorts to overly conservative and wasteful solutions. Recognizing this, there has been a wave of proposals for specialized memory hierarchies [2, 6–9, 23, 34, 36, 40, 50, 54, 58, 67, 75, 85, 90, 92, 95, 106–108, 118, 127, 131, 135, 136, 146, 149–154]. These designs are highly effective, often reporting speedups of 2× or more, so there is clearly potential to massively reduce data movement.

However, the elephant in the room is that adding custom logic to a general-purpose CPU memory hierarchy is very expensive. Taken literally, prior work suggests that memory hierarchies should contain an ever-growing number of custom accelerators. But this is unrealistic because each change to the hardware-software interface requires large, up-front investment in both hardware and software to be effective. Most accelerators benefit too few applications to justify such investment, creating innovation deadlock where large potential speedups cannot be realized in practice. Thus, optimizations are mostly limited to those that preserve the load-store interface, such as cache replacement policies or prefetchers.

We argue that the solution to this deadlock is to find a single, general-purpose architecture that supports a wide variety of data-movement features and optimizations. Only with wide applicability can the necessary hardware and software investment be justified. Additionally, we observe that the key to many prior optimizations is the ability to perform simple computations in response to data movement. Hence, the thesis of this paper is that: Architectures should expose more data movement to software, so that software can observe and optimize data movement itself. In other words, the hardware-software interface is the problem, and often specialized hardware is not needed with a richer interface. The missing ingredient is feedback from hardware to software when data...
moves. We call this idea a polymorphic cache hierarchy, and we propose the tākō\(^1\) architecture to realize it.

Software control of data movement offers enormous advantages over a hardware-only approach. Solutions can be better tailored to individual applications, and development cycles go from years to days. Although the upfront costs of a new hardware-software interface are formidable, **these costs are paid only once**, after which the marginal cost is reduced by orders of magnitude.

Fig. 1 illustrates tākō in action. Software (e.g., an application, domain-specific framework, or library) registers a **phantom address range** with tākō, whose data only lives in-cache and is not backed by off-chip memory [23]. Instead of fetching data from memory, misses to this address range are served by **software callbacks**. Evictions and writebacks are handled similarly. These callbacks thus define the semantics of loads and stores in this address range, letting software re-purpose the caches as desired.

Like recent near-data computing architectures [6, 83, 105, 142, 150], tākō adds programmable engines near caches to execute call-backs efficiently. In tākō, engines contain scheduling logic and a spatial dataflow fabric to run callbacks [43, 59, 103, 138, 143]. With this microarchitectural support, tākō gets close to the performance of fully specialized hardware — software programmability adds little overhead because data movement costs dominate and callbacks are short. The critical difference from prior work is that whereas cores invoke tasks in prior near-data architectures, **caches invoke callbacks in tākō**. This difference is the crux of the architecture: tākō closes the loop between hardware and software, letting software finally observe and optimize data movement.

This paper explores the programming interface and system architecture of a polymorphic cache hierarchy. tākō's goal is to enable optimizations that otherwise require custom hardware, and as such it currently provides a low-level interface for expert programmers. This paper focuses on (i) an initial set of callbacks that covers many, but not all, data-movement features and optimizations; and (ii) an architecture that implements these callbacks correctly and efficiently.

**Contributions.** This paper contributes the following:

- **Problem.** We identify the need for an improved hardware-software interface to unlock the performance and efficiency gains demonstrated by recent specialized cache hierarchies.
- **Programming Interface.** We propose a simple, flexible, and effective programming interface to give software visibility and control over data movement.
- **Architecture.** We discuss the architectural constraints and features needed to implement a polymorphic cache hierarchy correctly and with good performance, with similar hardware overhead to prior near-data computing architectures.

**Summary of results.** We present five case studies for tākō, demonstrating that a general-purpose, programmable data-movement architecture can enable new functionality while approaching the performance of custom hardware.

- **In-cache data transformation:** tākō enables software-defined transformations (e.g., decompression) when data moves. With good locality, tākō eliminates redundant work to get 2.2x speedup and 61% energy savings.
- **Commutative scatter-updates:** tākō implements PHI [95], transforming the caches to use push-based semantics to accelerate commutative scatter-updates in graphs. tākō gets 4.2x speedup, similar to [95].
- **Decoupled graph traversals:** tākō implements HATS [92] as a representative decoupled streaming application. tākō accelerates graph traversals and gets a 43% speedup and 17% energy savings.
- **Transactions on non-volatile memory:** tākō’s improved visibility over data movement eliminates wasteful work in NVM transactions. If no data is evicted before commit [91], tākō eliminates journaling overhead and achieves up to 2.1x speedup and 47% energy savings.
- **Detecting cache side-channel attacks:** tākō exposes data movement to software, letting applications detect and prevent cache side-channel attacks [81].

Unlike prior work that requires custom hardware for each feature and optimization, tākō implements these applications on a single, general-purpose hardware design. tākō adds just ∼5% area overhead, similar to prior near-data systems. Further, we show that tākō’s hardware achieves performance within 1.8% of an idealized design.

## 2 TĀKŌ OVERVIEW

Tākō consists of software and hardware components. In software, tākō’s programming interface gives software visibility and control over data movement via cache-triggered callbacks. In hardware, tākō adds architectural support for scheduling and executing callbacks efficiently near data.

**Design rationale.** Caches exist to shield systems from expensive operations. Traditionally, these are reads and writes to larger memories lower in the cache hierarchy, but in principle they could be anything. tākō opens up the cache hierarchy by letting software define what happens on a cache miss and, similarly, what to do with evictions.

Opening up the cache hierarchy yields two distinct benefits:

- (a) Software can leverage existing cache hardware to memoize expensive computations or buffer updates; and
- (b) Software can observe data movement as it happens and interpose as necessary.

Both of these benefits are essential to implementing many data movement features and optimizations. For example, PHI [95] (a) buffers graph updates in-cache, and (b) decides on eviction whether to apply updates in-place or log them (Sec. 8.1).

**Interface.** Table 1 summarizes tākō’s interface. Callbacks are registered only on selected addresses, and tākō does not affect loads and stores to other addresses. onMiss is invoked on cache misses, letting software fill in the requested cache line. Values are then cached normally; i.e., cores can read and write them, with hits handled like any other data. onEviction and onWriteback handle evictions for clean and dirty data, respectfully.

**Architecture.** Fig. 2 shows a high-level view of a tākō system. On top of a baseline, cache-coherent multicore, each tile is augmented...
with an engine that contains hardware scheduling logic and a programmable dataflow fabric to execute callbacks. täkō tracks which lines have callbacks registered and adds no latency or energy to traditional loads and stores.

The engine microarchitecture is guided by constraints and characteristics of täkō callbacks. To compete with specialized hardware, callbacks must exploit memory-level parallelism but should not add much area. Callbacks tend to be short, re-execute repeatedly, and perform the same operation across entire cache lines. These considerations led us to a dataflow fabric (to avoid re-fetching the same operations led us to a dataflow fabric to avoid re-fetching the same instructions) with SIMD functional units (for repeated operations).

Summary. täkō hardware enables visibility and control over data movement in software via its general-purpose programming interface. The architecture changes only once, up front, rather than for each individual data-movement feature or optimization. täkō thus massively reduces the barrier for optimizing data movement.

3 MOTIVATION

Memory hierarchies currently suffer from innovation deadlock: though specialization offers large benefits, it also requires prohibitively large, up-front investments in both hardware and software. Without strong demand from software, hardware vendors are reluctant to design, verify, and support new features; but without hardware support, software vendors will not rewrite applications. As a result, architects are limited to optimizations that preserve the load-store interface but leave significant gains on the table. The goal of täkō is to break this deadlock by providing a general-purpose architecture that frees software to optimize data movement itself.

To motivate täkō, we begin with an example of how polymorphic cache hierarchies enable data-movement optimization in software. The purpose of this example is to introduce the basic components of a polymorphic cache hierarchy. Later case studies will show the full power of polymorphic cache hierarchies to transform cache behavior.

### 3.1 Example program: Lossy compression

Prior work has studied many optimizations that transform data as it moves through the cache, e.g., to compress [9, 36, 90, 106, 107, 118, 136, 146], decrypt [47, 65, 115], prefetch [6, 131, 149], change layout [7, 23], memoize [8, 40, 153, 154], or serialize/de-serialize [108] data. We motivate täkō by observing how its onMiss callback enables arbitrary data transformations while improving performance, saving energy, and reducing overall work.

Fig. 3 shows our example program, which computes the average value of a data set that is stored in an approximate, compressed format in memory as a base plus offset value, similar to [107]. Unlike standard compressed caches, this lossy compression cannot be implemented in hardware without application knowledge [89], motivating the need for software in the loop. (The details of the compression algorithm are immaterial; the point is that software can transform data however it likes.)

This program has two major problems. Cores are inefficient at data transformations, wasting time and energy [108, 146]. And if data are re-used, then the program re-executes the same transformation many times. However, there is currently no good alternative in software, as alternative implementations waste memory, add data movement, or perform even more work.

3.2 täkō to the rescue

Fig. 4 illustrates how täkō solves these problems. Rather than operate on the raw compressed data, the program allocates a new “phantom” address range for decompressed data. These addresses only live in the caches and are not backed by physical memory. The program defines an onMiss callback that decompresses data whenever a new cache line in the phantom range is requested.

The callbacks are grouped in a Morph object that collects the data and methods for this polymorphic cache hierarchy — in this example, a data pointer to the phantom address range and pointers to the bases and deltas arrays. The onMiss callback takes the phantom address that triggered the miss and decompresses the requested data. All operations execute in parallel across the full cache line, shown in data-parallel pseudocode for brevity.

The modified program first registers the Morph at the private L2 cache, allocating a phantom address range for it. It then simply reads the decompressed data and computes the average, now using even simpler code. Fig. 5 illustrates its execution. The first time the program reads a phantom address X, there is an L2 miss, which triggers onMiss on the spatial dataflow engine to decompress the full cache line. The decompressed line is then cached so that any

### Table 1: täkō callback semantics.

<table>
<thead>
<tr>
<th>Callback</th>
<th>Semantics</th>
<th>Side effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>onMiss</td>
<td>Generates data for requested address.</td>
<td>✗</td>
</tr>
<tr>
<td>onEviction</td>
<td>Handles eviction of unmodified data.</td>
<td>✗</td>
</tr>
<tr>
<td>onWriteback</td>
<td>Handles eviction of modified data.</td>
<td>✓</td>
</tr>
</tbody>
</table>

* ✗ — can only write local state and/or the affected cache line; see Sec. 4.3.

### 1 Example program written in traditional software.

```plaintext
1 int64 bases[0 / 8] # one base per line, or 8 values
2 int8 deltas[8] # 4-bit exponent, 4-bit mantissa
3 total = 0
4 for idx in indices:
5 # 1. decompress data
6 base = bases[idx >> 3]
7 delta = deltas[idx]
8 mantissa = delta & 0b111
9 exponent = delta >> 4
10 data = base + (mantissa << exponent)
11 total += data
12 avg = total / len(indices)
```

Figure 3: Example program written in traditional software.
3.3 Results and comparison to prior work.

The täkō version of this program improves performance, saves energy, and reduces redundant work. Fig. 6 shows results with 32 K indices for the baseline software implementation, a software version that pre-computes the decompressed data in a separate array, a near-data computing (NDC) implementation, and the täkō version. The pre-compute version uses vector instructions to decompress a full cache line (eight values) at a time. The NDC version is similar to [83], where the core offloads decompressions to execute at an L2 engine. Indices are randomly generated following a Zipfian distribution [21] over 16 K values. (Full experimental methodology is in Sec. 7.)

Finally, while this example showed how täkō can leverage caches to eliminate redundant work, täkō is capable of more radical transformations of cache behavior. These will be explored in Sec. 8.

3.4 Discussion.

Decompression is representative of many prior optimizations that transform data as it moves through the cache hierarchy. Such transformations are easily implemented by writing onMiss, onEviction, and onWriteback callbacks. These callbacks are written in software and execute on täkō’s general-purpose hardware. Compared to adding custom hardware, täkō reduces the innovation barrier by orders of magnitude.

It bears emphasizing that a polymorphic cache hierarchy is not purely microarchitectural. This is by design: the entire point is to give software visibility and control over data movement. Callbacks should be thought of as part of the application code, which execute as hardware-scheduled threads in parallel with conventional software threads. A well-structured application splits functionality appropriately between the two.

Figure 4: Pseudocode for the same program in täkō.

```c
1 # define new cache semantics to pack data densely
2 class Decompressor extends takō::Morf:
3 int64 bases # actually in base Morph class
4 int64* deltas
5 # decompress data when it moves into cache
6 void onMiss(int64* phntmAddr):
7    idx = phntmAddr - &this.data[0]
8    base = bases[idx >> 3]
9    delta = deltas[idx]
10    mantissa = delta & 0b1111
11    exponent = delta >> 4
12    *phntmAddr = base + (mantissa << exponent)
```

Figure 5: Running the example program in täkō. Data is decompressed automatically on a cache miss, executing application onMiss code on a near-cache spatial dataflow fabric. Decompressed data is also cached for future use to eliminate redundant work.

![Figure 5](image-url)
Our goal is to massively reduce implementation effort vs. the custom hardware required by prior specialized cache hierarchies. This restriction simplifies translation hardware (see below), but it is not fundamental.

**4 TAKÔ PROGRAMMING INTERFACE**

takô’s programming interface is designed to let software optimize data movement in ways that would otherwise require custom hardware. Our goal is to massively reduce implementation effort vs. the custom hardware required by prior specialized cache hierarchies. This section describes the interface and restrictions that make it easier to reason about program behavior. Though takô is available to application programmers, it currently targets experts; we envision takô code being shipped as part of domain-specific frameworks or libraries.

**Overview.** takô breaks the address space into different address ranges, each with their own semantics. Software can register callbacks that execute in response to specific cache events—misses, evictions, and writebacks. By default, addresses retain load-store semantics and have no callbacks registered.

Software defines the behavior of a polymorphic cache hierarchy by providing a Morph data type and registering it with a specific address range. Often, the Morph allocates a new “phantom” address range that is not backed by physical off-chip memory [23], but Morphs can also be registered on “real” addresses. Phantom callbacks define the results of loads and stores to the address range, since there is no backing memory to load or store. Fig. 8 gives pseudocode for takô’s basic interface, discussed in detail below.

### 4.1 register/unregister.

Registering the Morph associates callbacks with an address range. Software provides a morph type (a child class of takô::Morph), the location in the cache hierarchy to register the Morph, and the address range. The location can be PRIVATE (at the L2) or SHARED (at the L3). Currently, takô does not support Morphs at the L1 because L1s are tightly integrated with cores; nor does it support Morphs at memory because memory controllers are below the cache coherence protocol, complicating consistency in callbacks.

Phantom address ranges are requested only by their size, and registerPhantom allocates and assigns the address range. To support Morphs on existing data, registerReal accepts an arbitrary base and bound and attempts to register the Morph on this range, takô only allows one Morph to be registered on an address at a time. This restriction simplifies translation hardware (see below), but it is not fundamental.

The Morph remains in effect until unregistered. When a Morph is registered or unregistered, its address range is flushed from the cache. unregister de-allocates phantom address ranges.

### 4.2 Morph objects.

A Morph object represents an instance of a particular polymorphic cache hierarchy. Multiple instances of a Morph type, or of different types, can be registered at the same time, each operating on their own distinct address ranges (e.g., see Sec. 8.3). register returns a Morph object, letting software threads control it (e.g., by unregistering it).

Callbacks execute on engines, not cores, and each engine also has its own view (i.e., copy) of the Morph object. This is important because each view may have local state, similar to conventional thread-local state, but shared by all threads running on that engine. Local state is allocated in memory, and engines access it via coherent loads and stores. PRIVATE Morphs have a single view (at the L2), but SHARED Morphs have one view per L3 bank. The views are gathered in the views array to, e.g., allow initialization of local state.

### 4.3 Callbacks.

Cache-triggered callbacks are the heart of takô’s design. By defining callbacks in the Morph, software transforms the semantics of that address range. Callbacks are flexible to maximize takô’s applicability, but they must obey certain restrictions for correctness and performance.

**Semantics.** takô callbacks allow software to modify cache behavior, as summarized in Table 1. For phantom address ranges, onMiss and onWriteback directly define the results of loads and stores. When there is a miss to a phantom address, the cache controller allocates a line, zeroes it, and then invokes onMiss. When evicting a phantom cache line, the cache controller invokes onEviction (if clean) or onWriteback (if dirty) and then discards the line. Inter-vening memory operations (i.e., cache hits) simply read and write the data normally, without invoking callbacks.

Callbacks on real address ranges operate similarly, except that the cache controller reads and writes the backing memory, maintaining load-store semantics by default. onMiss begins executing in parallel with reading addr. onWriteback executes before writing back addr to let the callback interpose.

onMiss is on the critical path of software threads, but onEviction and onWriteback are not. This difference is important for performance: it is best to keep onMiss short, and push work into the other callbacks (e.g., see Sec. 8.1).

**Execution model.** Callbacks are short threads that are created and scheduled entirely by hardware and run in parallel with conventional software threads (Fig. 9). Because callbacks are triggered by cache hardware, they can occur spontaneously from the perspective of a software thread. This spontaneity can be unintuitive: cache misses can be triggered by speculative loads or prefetches, so an onMiss may not correspond to any committed instruction in a program. Similarly, data can be evicted from caches at any time, triggering onWriteback even when no corresponding software thread is active.

**Restrictions.** Given these considerations, it is best practice to write callbacks that behave similarly to conventional reads, evictions, and writebacks. That is, onMiss and onEviction should
be free of side effects, since they can be triggered at any time, whereas onWriteback can have side effects, since modified data must correspond to a committed store in some software thread. These restrictions make it easier to reason about callback behavior, but tâkô does not strictly enforce them because misses/evictions are sometimes part of correctness (e.g., for security; see Sec. 8.4).

Ignoring side effects, callbacks can reference nearly any memory address. The remaining exception is that callbacks cannot access data with a Morph registered at the same or higher level of the cache hierarchy (Fig. 9). Without this restriction, deadlock is possible as callbacks trigger further callbacks, quickly exhausting the engine’s hardware scheduler. A SHARED callback is not allowed to trigger a PRIVATE callback because the PRIVATE callback could trigger onMiss in the shared cache. But a PRIVATE callback can trigger a SHARED callback, since there is no cyclic dependence. This constraint was not problematic in any of our case studies.

Callback code. tâkô is designed for short callbacks, which we find to be natural in our case studies. Callback code executes in SIMD fashion across entire cache lines. For long code paths or error conditions, callbacks can raise a user-space interrupt to preempt a software thread (e.g., see Sec. 8.4). For simulation convenience, callback code is currently written in C++, and instructions are mapped onto the datalow fabric when they first execute; in practice, one could compile code statically [130, 143].

Coherence and consistency. tâkô leverages the cache-coherence protocol in the baseline multicore to provide a consistent view of memory. A callback is just another thread in the system, from a consistency perspective. Engines have coherent L1d caches, implemented using clustered coherence within each tile to avoid increasing directory state [49, 77, 88]. In brief, the L2 and engine L1d snoop on coherence traffic within each tile so that the directory behaves exactly as if the engine L1d is part of the L2 cache on that tile.

Callbacks thus enjoy the same coherence and consistency as any other thread in the system. Additionally, the address that triggered the callback is locked for the duration of callback execution; i.e., no other thread (or callback) can access the data until the callback completes. Locking is strictly enforced by the cache controller, which serializes operations on each address. Callbacks therefore do not need to worry about racing accesses to addr, but races to other addresses are possible, so callbacks should be data-race free [1] to maintain consistency.

4.4 flushData. flushData enables synchronization between callbacks and conventional threads without completely unregistering a Morph. By flushing all of a Morph’s data from the cache, programs are guaranteed that there will be no further racing writes from callbacks. flushData signals cache controllers at the appropriate level of the hierarchy to walk their tag arrays and flush any lines belonging to the Morph’s address range, triggering onWriteback or onEviction. flushData blocks the software thread until all callbacks complete.

4.5 Discussion and roads not taken.

We found the above callbacks to be a logical starting point for a polymorphic cache hierarchy that covers a wide range of use cases. As discussed in Sec. 2, the basic intuition is to generalize caches by letting software provide an onMiss handler [56], and the rest of the interface and its restrictions follow naturally. We arrived at this interface early in the design, and it proved useful, self-contained, and consistent. Although the semantics are not trivial, writing tâkô software has been fairly straightforward in our experience. For most applications, there is a clear separation of concerns across misses and clean or dirty evictions (Sec. 8).

That said, more callbacks are certainly possible. onReplacement would allow software to optimize the eviction policy for particular workloads [10, 145]. onHit would allow customization of the cache coherence protocol, among other applications. We did not pursue onHit because programmable cache coherence has been explored extensively [3, 23, 34, 73, 75, 113, 123, 151, 152] and because it seemed that onHit would often be needed in the L1, requiring disruptive core changes. Finally, one could make cache indexing programmable, letting software re-purpose the tag array [111, 112, 116, 121, 122, 154]. We did not explore this direction to avoid adding any latency to conventional loads and stores — tâkô has no performance impact on legacy applications.

5 TâKÔ ARCHITECTURE

Similar to recent near-data-computing architectures [6, 83, 105, 142, 150], tâkô extends a baseline multicore with near-cache engines to run callbacks efficiently (Fig. 2). Engines are placed on each tile of the multicore, near the L2 and L3 caches. The engines consist of (i) a hardware scheduler that buffers callbacks and runs them when they are ready, and (ii) a spatial dataflow fabric that executes callbacks efficiently. Fig. 10 shows onMiss and onWriteback callbacks, which are referenced throughout the text below.

5.1 Core modifications for tâkô.

Tracking Morphs. tâkô tracks which addresses have a registered Morph via the TLB. TLBs are augmented with two bits indicating whether a Morph is registered and, if so, whether it is registered at PRIVATE or SHARED. When a load or store misses, the core augments the GET request with these bits, giving the Morph’s location 3. Alternatively, tâkô could keep a separate table of registered Morphs, but this would limit the number of Morphs that could be registered concurrently.

Figure 9: Callbacks are scheduled by hardware in response to cache misses, evictions, and writebacks, and run on the engine closest to the data. Only writebacks should have side effects.
SA. tâkô adds one new cache flush instruction, corresponding to the flushData API, that flushes a particular address range in the PRIVATE or SHARED cache.

### 5.2 Cache modifications for tâkô.

**State.** Tags are extended with one bit to track whether a Morph is registered for the line at that cache level. This bit is set on insertion using the two registration bits in the GET request.

**Triggering a callback.** Engines are tightly integrated with the cache controller. When serving a cache miss, eviction, or writeback, the controller checks whether a Morph is registered and, if so, sends a request to the local engine along with the addr and operation type. The engine’s scheduler enqueues a request in its callback buffer, which starts executing it as soon as the fabric is available and the callback configuration is loaded. (Usually, the fabric is ready immediately.) For onEviction and onWriteback, the registered line occupies an entry in the cache’s writeback buffer until a callback buffer entry is available. When the callback completes, the cache controller responds to the original request. Other cache operations (i.e., all hits and any operation with no Morph registered) work normally and do not go through the engines at all.

**Avoiding deadlock.** Without additional mechanisms, deadlock can occur in the engine scheduler: e.g., suppose the engine’s callback buffer is full, an executing callback suffers a cache miss, and every line in the set is waiting to grab a callback buffer spot (e.g., to execute onMiss). Nothing can be evicted because the callback buffer is full, so the callback buffer cannot drain.

Luckily, it is easy to avoid this deadlock by ensuring that there is always a cache line in every set with no Morph registered at this cache or any child cache. This constraint guarantees forward progress, as there will always be a line that can be evicted without triggering a callback. tâkô enforces this constraint by modifying its eviction policy, tripp (see below). For similar reasons, tâkô enforces that there is always at least one MSHR and writeback buffer entry not waiting on a callback.

**Avoiding cache pollution from callbacks.** Callbacks often translate a phantom address to some real address that is accessed during the callback, but is not accessed afterwards (e.g., deltas[idx] in Fig. 4). To avoid cache pollution, tâkô modifies its RRIP-based [62] replacement policy, tripp, to insert accesses from engines at lower priority (i.e., closer to eviction). This optimization can significantly improve cache utilization; e.g., in a simple Morph that maps array-of-structs to struct-of-arrays, we have observed speedup of $>4\times$.

### 5.3 Engine microarchitecture.

tâkô adds one engine to each tile of the CMP. The engine runs all callbacks for the L2 and L3 bank on that tile. It has its own cache-coherent L1 data cache, a small TLB and reverse TLB for address translation, and a spatial dataflow engine to execute callbacks.

**Scheduling callbacks in hardware.** The scheduler consists of simple logic in hardware and a buffer of pending requests. Upon receiving a callback request, the engine enqueues it in its callback buffer, assigns the callback a unique id, and loads the callback bitstream into the fabric (if necessary). The engine maintains a small bitstream cache, which maps Morphs’ registered address ranges to their callbacks’ bitstreams and tracks which callbacks are loaded on the fabric. Callbacks begin executing once the fabric is ready and all earlier callbacks on the same addr have finished.

**Dataflow fabric.** Callbacks execute on a small dataflow fabric; see Fig. 11. The fabric is an array of simple processing elements (PEs) connected by an on-chip network. Each PE contains an instruction memory that holds a small number (e.g., 16) of static instructions, a token store that holds intermediate values, and ALUs. PEs issue operations using asynchronous dataflow firing, supporting concurrently executing callbacks via dynamic tag matching [59, 103, 138, 143] on callback ids. Operations work in SIMD fashion across entire cache lines at a time.

Our workloads require only a small fabric (e.g., $5\times5$ with simple integer operations and few (e.g., 8) concurrent callbacks (see Sec. 9). Our largest Morph, for HATS (Sec. 8.2), contains 94 instructions across all its callbacks, less than one-quarter of fabric resources. Our next-largest application contains only 46 instructions. Moreover, across all applications, there are no more than 19 average live tokens when an engine is active (summing across concurrent callbacks). There is thus plenty of room for co-running applications to share engines, even without mechanisms to limit contention (see Sec. 6).

We chose dataflow fabrics for tâkô engines because (i) callbacks are typically short, (ii) callbacks are frequently executed in parallel,
and (iii) callbacks are executed repeatedly. Short callbacks map easily onto a small, dynamic dataflow fabric, letting tākō run callbacks near-data with low area overhead. A dataflow fabric can easily run callbacks in parallel by assigning each a unique tag. Alternatively, tākō could execute callbacks on reserved SMT threads [141, 151], but this would either sequentialize callbacks or require multiple, heavy-weight thread contexts. Moreover, constantly re-fetching and decoding the same instructions would be wasteful. Preliminary exploration of SMT threads showed severe performance penalties, and Sec. 9 finds that in-order cores, as proposed in prior work [6, 83], perform very poorly in tākō.

5.4 Putting it all together.

Tākō’s hardware support adds little area to the baseline multicore system (Table 2). With 512 KB L3 banks and 64 B lines, the L3 tags need 1 KB to track Morph registration. The engines have 8 KB L1d caches, 2 KB TLB and rTLBs (see below), and a 5 × 5 dataflow fabric with integer functional units. Conservatively overprovisioning the token and instruction memory yields state overhead of 5.3% over an L3 bank. This is comparable to recent fabrics [97, 114, 143], which add roughly 5% area overhead.

6 SYSTEM INTEGRATION

By opening up the cache hierarchy to software, tākō touches many aspects of the system stack. This paper does not solve every issue, but here we discuss some of the major implications of polymorphic cache hierarchies.

Address translation. Caches use physical addresses, but tākō callbacks need virtual addresses. The engines maintain a reverse TLB (rTLB) for this purpose. The rTLB is eagerly filled when an onMiss is scheduled 3; however, we found that this optimization makes little difference in our workloads because rTLB hit ratios are so high. When a callback is scheduled, the engine recovers the virtual addr using the rTLB and the physical address from the cache tags 4b. Synonyms (i.e., ambiguity in reverse translation) are not an issue because only one Morph can be registered on an address at a time. The engine also keeps a conventional L1 TLB for other data accessed by callbacks, sharing the L2 TLB with the main core.

Tākō has several nice features with respect to address translation. Phantom addresses are not backed by physical memory, making huge pages easier to use because fragmentation is less of a concern than in conventional memory allocators [74]. Moreover, the engines’ rEviction and rWriteback can only be triggered on cached data. Both of these observations mean that the engine rTLB can be small (Sec. 9). We assume that engine TLBs are kept coherent using shootdowns when translations change (e.g., when a Morph is registered or unregistered).

<table>
<thead>
<tr>
<th>Table 2: Hardware overhead (state per L3 bank).</th>
</tr>
</thead>
<tbody>
<tr>
<td>L3 tags</td>
</tr>
<tr>
<td>Engine L1d, TLB, rTLB</td>
</tr>
<tr>
<td>Callback buffer</td>
</tr>
<tr>
<td>Token store</td>
</tr>
<tr>
<td>Instruction Memory</td>
</tr>
<tr>
<td>Total per L3 bank</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3: System parameters in our experimental evaluation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
</tr>
<tr>
<td>Engines</td>
</tr>
<tr>
<td>L1</td>
</tr>
<tr>
<td>L2</td>
</tr>
<tr>
<td>LLC</td>
</tr>
<tr>
<td>NoC</td>
</tr>
<tr>
<td>Memory</td>
</tr>
</tbody>
</table>

OS support. Tākō requires operating system support to manage Morph registration. The operating system needs to track which address ranges currently have a Morph registered along with a pointer to the callback code. Phantom address ranges may require an independent data structure from the page tables, since they use physical addresses that do not correspond to physical memory. Morphs also complicate thread scheduling because eviction callbacks can still run even if a process is de-scheduled from cores. In many cases, this is not problematic. But if a process must be fully de-scheduled for some reason, then it is necessary to also flush its Morphs’ data (i.e., using the flushData API). Doing this is feasible but takes time and energy, especially for Morphs at the \( \text{SHARED} \) cache.

Multi-tenancy, virtualization, and security. In heavily shared systems with many active Morphs, further potential problems arise with thrashing in engines, possible security issues between concurrent callbacks, and virtualizing shared resources. These issues are outside the scope of this paper, but we think partitioning application data across L3 banks is a promising solution [100, 120]. That is, the operating system can prevent unwanted contention or interaction between callbacks by preventing them from sharing cache space in the first place.

7 EXPERIMENTAL METHODOLOGY

Simulation framework. We evaluate tākō in execution-driven microarchitectural simulation. Our simulator shares infrastructure with SwarmSim [63], but supports cycle-level timing throughout the memory hierarchy and models tākō’s interface and engines.

System parameters. Except where specified otherwise, our system parameters are given in Table 3. We model a tiled multicore system with 16 cores connected in a mesh on-chip network. Each tile contains a conventional out-of-order core (modeled after Intel Goldmont), one bank of the shared LLC, and a tākō engine. Sec. 9 varies these parameters and shows that tākō is effective across a variety of system configurations.

We assume the out-of-order cores support atomic exchange operations (e.g., LL/SC) along with other relaxed atomics. Except where noted, we evaluate engines with a 5 × 5 dataflow fabric (15 integer PEs and 10 memory PEs) with 1-cycle PE latency. We also evaluate an idealized engine with unlimited, 0-cycle latency PEs; i.e., callback latency is only affected by memory latency and data dependencies.

Metrics. We present results for speedup and dynamic execution energy (energy parameters from [114, 133]). We focus on dynamic energy because tākō has negligible impact on static power and to clearly distinguish tākō’s impact on data movement energy from its overall performance benefits.
8 EVALUATION — CASE STUDIES ON TÄKÖ

Täkô’s flexible programming interface enables a wide variety of optimizations on the same, general-purpose hardware. We evaluate a sample of four applications that can benefit from täkô to demonstrate:

- Täkô supports prior specialized cache hierarchies. We implement two prior designs that accelerate graphs in very different ways [92, 95].
- Täkô enables features in software that are impossible without fine-grain visibility over data movement. Specifically, täkô lets the system eliminate unnecessary writes in direct-access NVM and detect suspicious activity.
- Täkô’s performance is fairly insensitive to its microarchitectural parameters (Sec. 9) and close to an idealized design.

Our case studies depend on being able to observe and interpose on data movement, and are thus not implementable on prior near-data computing (NDC) architectures. Täkô provides the missing interface and mechanisms to implement these data-movement optimizations in software.

8.1 Accelerating commutative scatter-updates.

We begin with an example of how täkô can redefine cache semantics to accelerate data movement. This study implements PHI [95], a push-based hierarchy for commutative scatter-updates, e.g., in graph applications. PHI turns the cache into a large write-combining buffer for commutative operations (e.g., addition). In PHI, the cache contains updates (e.g., deltas), not raw data. When a cache line is evicted, PHI either immediately applies the update in-place or logs the update to be applied later [14, 70]. PHI minimizes memory bandwidth by choosing between these two policies, using the number of updates in the line to decide which is best.

Description. Fig. 12 illustrates how täkô implements PHI. The application starts by allocating a phantom address range the same size as the graph’s vertex data. In the first phase, updates are pushed to the phantom region using remote memory operations (RMO) (i.e., relaxed atomic add [126]). If updates miss in the cache, they trigger onMiss to initialize the lines with an identity element (e.g., zero for addition), without making any requests down the cache hierarchy. The application then pushes commutative updates to the cache (i.e., write hits). When a line is evicted from the cache, onWriteback either directly applies the updates to backing memory or appends them to a “bin” depending on the number of non-identity values in the line. After completing the edge phase, the main thread calls flushData and then streams through the bins to apply deferred updates (not shown).

Why täkô? PHI’s design fits very well with täkô’s interface. Its implementation requires application- and data-dependent optimizations on cache lines as they are allocated and evicted. This is exactly the type of data-movement control that täkô enables in software. Moreover, PHI is a prime example of the limitations of prior NDC: PHI requires the ability to intercept misses and writebacks and modify their behavior, which is not possible in traditional NDC.

Table 4: täkô callbacks for PHI.

<table>
<thead>
<tr>
<th>Callback</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>onMiss</td>
<td>Sets line to identity element (e.g., zero).</td>
</tr>
<tr>
<td>onEviction</td>
<td>—</td>
</tr>
<tr>
<td>onWriteback</td>
<td>If # updates &gt; threshold, apply updates immedi-</td>
</tr>
<tr>
<td></td>
<td>ately; otherwise, log updates for application</td>
</tr>
<tr>
<td></td>
<td>in “binning” phase.</td>
</tr>
</tbody>
</table>

![Figure 12](image-url)  
**Figure 12:** Täkô lets software re-purpose the cache to accelerate applications. PHI accelerates scatter-updates by buffering updates in-cache and applying them when evicted. Writebacks either apply updates in-place or log updates to be applied later. These optimizations are naturally implemented in täkô via onMiss and onWriteback.

![Figure 13](image-url)  
**Figure 13:** PHI results for PageRank on a 16M vertex, 160M edge synthetic graph. Täkô improves performance by 4.2×.

**Evaluation.** Fig. 13 shows results for PageRank with 16 threads pushing updates to a single Morph registered at SHARED,3 comparing täkô to a baseline software implementation, a software implementation of update batching (UB) [14, 70], and an ideal dataflow engine. We see similar results as the PHI paper [95]: UB in software gets 3.2× speedup, but täkô gets 4.2× speedup. Täkô also reduces energy by 36%, compared to 27% for UB.

Täkô achieves its benefits by (i) writing to phantom data, which does not incur a memory access on miss; (ii) binning updates off the critical path of the main threads on writeback; and (iii) reducing memory accesses and core computation compared to UB (by 29% each) by buffering updates in the cache and sometimes applying them in-place. Fig. 14 breaks down memory accesses for each implementation between the edge, bin, and vertex phases of PageRank. UB reduces total accesses by 43% by improving spatial locality via binning. Täkô reduces total accesses by 60% by buffering updates in-cache and only binning when

3Due to simulator limitations, we can currently only run PHI at a single level. But täkô’s design allows hierarchical PHI as described in [95], which would show even better results.
there is poor spatial locality, lowering accesses in both edge and bin phases. Further, tākō incurs negligible overheads compared to an ideal engine because onWriteback is short (35 cycles and 21 instructions on average), off the critical-path, and most of the latency comes from memory accesses (0.17 accesses per onWriteback on average).

8.2 Accelerating graph traversals via streams.

This second study takes a much different view of accelerating graph applications by using tākō to implement a programmable, decoupled stream. Architectures have long had special support for streaming access patterns [31, 35, 45, 69, 99, 139, 140, 142], many of which use dedicated engines to stream data to the main cores. We demonstrate tākō’s support for programmable streams by implementing HATS (hardware-accelerated traversal scheduling) [92], which computes an efficient graph traversal to improve data locality in graph applications.

Description. HATS observed that, without expensive pre-processing, it is inefficient to process edges in the order they are laid out in memory. Many graphs exhibit strong community structure [13, 78], so it is much better to process graphs one community at a time. A bounded, depth-first search (BDFS) is a simple traversal order that significantly improves locality. The challenge is that BDFS is a poor fit for cores due to unpredictable control flow, so HATS adds a dedicated hardware engine.

Fig. 15 illustrates the tākō implementation of HATS. The application initially allocates a phantom address range large enough to hold every edge of the graph (recall that no physical memory is allocated). This phantom address range acts as a stream, where the core reads edges sequentially and the engine supplies edges when requested by onMiss. HATS’s onMiss keeps a small stack and walks the graph in BDFS order, as described in the original paper [92]. Our current implementation of HATS sequentializes all onMisses to simplify contention on the shared stack. While the core processes one part of the stream, the prefetcher triggers onMiss for subsequent edges. Note that onMiss is not guaranteed to be called in strictly sequential order, but this is fine in HATS because minor re-orderings have minimal impact on locality.

However, a more serious concern is that phantom lines can be evicted before the core has processed them. Although this occurs exceedingly rarely, the application cannot tolerate any lost edges. tākō solves this problem by logging unprocessed edges to memory in onWriteback and onEviction. To know which edges have been processed, the core assigns an INVALID value to processed edges using an atomic exchange (e.g., LL/SC). Any unprocessed edges are logged during onWriteback and onEviction, and the core processes the logged edges at the end of the iteration.

Why tākō? HATS is a good example of a streaming computation that runs inefficiently on cores, motivating the need for separate streaming hardware [6, 142, 150]. This case study shows how tākō can support this important class of workloads. For performance, HATS relies on decoupling between graph traversal (on engines) and edge processing (on cores); this is awkward if not impossible to implement in NDC. Moreover, implementing HATS in tākō software

![Figure 15: tākō’s stream implementation supports complex decoupled pipelines. HATS improves locality in graphs by traversing the graph in bounded depth-first order so that communities are visited together. onMiss provides a simple, data-movement triggered approach to filling a stream.](image)

Table 5: tākō callbacks for HATS.

<table>
<thead>
<tr>
<th>Callback</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>onMiss</td>
<td>Fills line with edges in BDFS order.</td>
</tr>
<tr>
<td>onEviction</td>
<td>Logs unprocessed edges.</td>
</tr>
<tr>
<td>onWriteback</td>
<td>Logs unprocessed edges.</td>
</tr>
</tbody>
</table>

![Figure 16: HATS results for one iteration of PageRank on uk-2002 graph [33]. tākō improves performance by 43% and reduces energy by 17% vs. the software baseline.](image)

Figure 17: HATS performance breakdown. Left: DRAM accesses split by PageRank phase. Mid: core branch mispredictions per graph edge processed. Right: cumulative core load latency.

lets it support a wide range of graph data formats or traversal heuristics [22, 94], unlike fixed-function hardware.

Evaluation. Fig. 16 presents speedup and energy results for a single thread of PageRank with the baseline “vertex-ordered” traversal, a baseline software BDFS implementation, tākō, and an ideal engine. Baseline BDFS provides minimal benefits due to extra instructions with complex control flow. In contrast, tākō provides substantial speedup of 43%, approaching the 46% speedup of an ideal engine. tākō also reduces energy by 17%, compared to 22% for ideal.

This speedup is due to (i) better cache locality; (ii) regularizing control flow on the core; and (iii) decoupling edge traversal from the core. Fig. 17 quantifies these points. All versions incur the same number of accesses during the vertex phase, but the BDFS traversal (also used by tākō) reduces misses to vertex data during the edge
phase by 40%. tākō also eliminates branch mispredictions by turning the complex BFS traversal into simple loop over a sequential stream, whereas software BDFS increases mispredictions by 52%. Finally, tākō reduces memory latency seen at the core by 19% over BDFS by decoupling the edge traversal.

tākō achieves significant speedups on HATS, but somewhat lower than reported in [92]. This is because we sequentialize the calls to onMiss, whereas [92] re-orders the trace to exploit locality by traversing multiple neighbors in parallel and processing whichever data returns first.

8.3 System support: Transactions in direct-access NVM.

We next show how better visibility over data movement enables new features and optimizations. There are many applications where it would be useful to know when data moves in or out of caches: e.g., for immutable data structures [19], intermittent computing [28, 84, 86, 87], checking data integrity [67, 144, 156], debugging and logging [25, 91], etc. This study considers a filesystem on non-volatile memory (NVM) with battery-backed caches, like Intel eADR [60]. The major challenge is to avoid inconsistent states on failure. For this purpose, NVM filesystems employ transactions using journaling, logging, or shadow paging [144, 156].

Description. Fig. 18 illustrates efficient journal-based transactions in tākō. Like prior transactional memory designs [91], the idea is that if a transaction’s writes complete before any have been evicted from cache, then it is safe to push the updates directly to NVM without journaling. (In a sense, the cache is the journal.) The application writes all updates to a phantom address range 3. To commit a transaction, the thread simply flushes the morph’s phantom data from the cache 4. onWriteback either writes directly to NVM (if the transaction has committed) or journals the writes (if not) 5. In the common case where no data is evicted, tākō adds minimal overhead 3. But if data is evicted before commit, then the application must apply the journaled writes to commit the transaction 3. This design permits one in-flight transaction per morph instance, but an application can register many instances. We register at PRIVATE because each transaction needs to flush all the phantom data, which is more efficient in the L2.

Why tākō? Current NVM filesystems must implement transactions conservatively because they cannot observe when data enters or leaves caches. Journaling avoids writing directly to data, but adds instructions and NVM writes. tākō lets filesystems only resort to journaling if data falls out of the cache. Prior NDC systems do not improve visibility over data movement and hence do not enable this feature.

Evaluation. Fig. 19 shows results for a workload of append-only transactions of different sizes, from 1 KB to 128 KB. As long as transactions fit in the L2, tākō provides up to 2.1× speedup by eliminating unnecessary journaling. tākō executes ≈50% fewer core instructions and ≈36% fewer total instructions (Fig. 20), yielding large speedup and up to 47% energy savings. tākō achieves the same gains as the ideal engine because the engine mainly performs very simple data copies. When the transaction size exceeds the cache size (i.e., 128 KB), onWriteback falls back to journaling and performs closer to the baseline. However, tākō still outperforms the baseline by filling the journal in onWriteback, off the critical-path of the core.

8.4 Detecting side-channel attacks.

Finally, we demonstrate tākō’s security benefits by showing how it can defend against prime+probe attacks [81] at the shared cache. This study emphasizes the additional functionality enabled by better visibility over data movement. Specifically, we demonstrate that tākō enables fine-grain monitoring of data for side-channel attacks [12, 48, 52, 61, 68, 79, 81, 96, 125, 147].

Threat model. We consider a scenario with attacker and victim threads running on separate cores in a CMP with shared last-level cache. The attacker detects when the victim accesses a vulnerable data structure (e.g., AES tables) to reverse engineer secure data (e.g., AES keys). We consider a prime+probe attack, but prior work has used similar techniques to defend flush+reload, evict+time, and cache+collision [26, 46].

Description. The prime+probe attack [81] leaks information about a victim process simply by detecting which cache set the victim

---

**Table 6: tākō callbacks for NVM support.**

<table>
<thead>
<tr>
<th>Callback</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>onMiss</td>
<td>Sets line with INVALID value.</td>
</tr>
<tr>
<td>onEviction</td>
<td>—</td>
</tr>
<tr>
<td>onWriteback</td>
<td>If transaction committed, apply writes immediately; otherwise, journal writes.</td>
</tr>
</tbody>
</table>

---

**Figure 19:** Results for NVM journaling microbenchmark at different transaction sizes. tākō improves performance by up to 2.1× and reduces energy by up to 47%.

**Figure 20: Instructions executed for each 8B written in application.**
accesses, as shown in Fig. 21a. The attacker starts by priming a target cache set with its own data. After the victim has accessed its secure data, the attacker then monitors how long it takes to probe its own data. Long latency (due to cache misses) reveals to the attacker which sets the victim has accessed, and thus leaks the victim’s access pattern. This prime-probe attack has been shown to leak entire AES keys [48, 61, 79].

täkō gives the victim visibility over movement of their secure data. Specifically, to detect a prime-probe attack, the victim needs to know when data is evicted. The application registers a “real data” Morph for the address range of its secure data (e.g., AES tables). The Morph only implements one callback, onEviction, which simply interrupts the main thread whenever any cache line containing the AES tables is evicted. This interrupt lets the victim defend itself from attack [12, 102, 125]. Fig. 21b shows a cache-eviction trace of an attack that is successful without täkō (left) and unsuccessful with täkō (right). täkō interrupts the victim during the probe phase of the attack before any information is leaked.

Why täkō? täkō exposes software to previously invisible data movement. Although active attackers can time cache accesses to expose microarchitectural state, passive victims might never even know they were attacked. täkō provides victim applications the tools to monitor data movement for cache attacks. This allows victims to take control over their data and defend themselves proactively. Like transactions above, visibility over data movement is the key to this defense, and prior NDC systems offer no solution.

9 EVALUATION — SENSITIVITY STUDIES

Engine microarchitecture. We study täkō’s sensitivity to engine microarchitecture on HATS. HATS is most sensitive to the fabric because its onMiss is the longest callback among our benchmarks. Fig. 22 evaluates different dataflow-fabric sizes, as well as an inorder core and ideal. Dataflow vastly outperforms inorder, but performance plateaus with small fabrics. We use a 5 × 5 fabric, which is within 1.8% of ideal. Fig. 23 evaluates HATS on a 5 × 5 fabric, varying arithmetic PE execution latency. We use single-cycle latency, but even at eight cycles speedup only decreases to 30% from 43%. This is because memory-level parallelism, not arithmetic throughput, is what matters most for täkō (Sec. 5.3).

Core microarchitecture. Fig. 24 evaluates PageRank on PHI with different core microarchitectures. Speedup is unchanged because PageRank is memory-bound. Beefier cores improve performance in absolute terms on decompression and HATS, but täkō’s speedup is affected little.

Scalability. Fig. 25 evaluates PageRank on PHI across different system and data sizes. (Memory bandwidth scales proportionally with cores.) täkō consistently outperforms update batching and improves with data size. täkō outperforms update batching by ≈34%, 32%, and 21% at 8, 16, and 36 cores, respectively. Hierarchical PHI would improve PHI’s speedup further at larger core counts by reducing cross-chip coherence traffic.

Callback-buffer size. The NVM journaling benchmark invokes many concurrent onWritebacks when flushing data, stressing the callback buffer. Varying the callback buffer from 1 to 64 entries, performance plateaus at 4 entries. Accordingly, we use 8 entries as a practical but sufficient size in our evaluation.

rTLB size. Finally, we swept rTLB size from 256 to 1024 entries with both 4 KB and 2 MB pages, and found that performance varied by at most 2.1%. We use 256 entries with 2 MB pages.
10 RELATED WORK

The cost of data movement. Data movement is more expensive than compute and only growing more so \cite{30, 53, 55, 76}. Even with inefficient out-of-order cores, data movement often consumes the majority of execution time and energy. Architectural specialization is no panacea: specialization makes data movement relatively more expensive \cite{32, 38}, and a significant fraction of programs will always run on general-purpose cores \cite{119}. Architectures simply must become more efficient at data movement.

Specialized cache hierarchies. These trends have been widely recognized, and there are many proposals to accelerate data movement, e.g., in machine learning \cite{2, 50}, graph analytics \cite{92, 95, 150}, data structures \cite{54, 58, 154}, memoization \cite{8, 40, 153, 154}, compression \cite{9, 36, 90, 106, 107, 118, 136, 146}, data layout \cite{7, 23, 155}, prefetching \cite{6, 131}, coherence and synchronization \cite{34, 75, 151, 152}, memory management \cite{85, 135}, and system software \cite{67, 108, 127}. While highly effective, they share the drawback of requiring custom hardware.

Software control of data movement. There has been some work that attempts to give software more control over the cache through better hardware partitioning mechanisms \cite{15, 29, 37, 39, 110, 117, 118}, software policies \cite{16, 17, 24, 66, 82, 120}, or a richer interface \cite{93, 137}. These works are complementary to tākö: they control data movement behind the load-store interface, whereas tākö expands that interface.

Near-data computing. Rather than move data to compute, some architectures move compute to data. Many of these designs are discrete “processing in-memory” co-processors that integrate logic in memory \cite{27, 44, 64, 71, 80, 98, 101, 104, 128, 133}, high-bandwidth memory \cite{7, 11, 18, 20, 41, 42, 57, 109, 148, 157}, co-processor designs make sense on streaming applications, but they are ill-suited to applications with significant data reuse or fine-grain communication \cite{5, 57, 83, 143, 144}. Other architectures enable near-data computing within a CPU’s memory hierarchy, letting cores offload work to memory \cite{5, 51} or caches \cite{2, 5, 83, 105, 129, 142}. However, there is no mechanism to trigger software when data moves, which we have shown is essential to many data-movement optimizations. tākö provides this missing mechanism.

Programmable memory hierarchies. Finally, the most related work is prior programmable memory hierarchies. The first programmable memory hierarchies were explored in the ’90s and focused on distributed cache coherence \cite{3, 23, 56, 73, 113, 123}. More recently, designs have added some programmability to the memory hierarchy for specific purposes: e.g., prefetching \cite{6, 131} or compression \cite{146}. By contrast, tākö targets a much wider set of features and optimizations by providing a general-purpose interface and architecture to increase software’s visibility and control over data movement.

11 CONCLUSION AND FUTURE WORK

Many inefficiencies in current systems are the result of an outdated hardware-software interface that gives software too little visibility and control over data movement. Polymorphic cache hierarchies expand the hardware-software interface to expose more data movement to software. tākö is an efficient, general-purpose implementation of a polymorphic cache hierarchy that massively reduces the innovation barrier for data movement features and optimizations. We demonstrated the wide applicability of tākö in five case studies.

Polymorphic cache hierarchies open up several exciting directions for further research. The current programming interface is low-level and intended for experts as an alternative to custom hardware. Language and compiler support would make polymorphic cache hierarchies more approachable for programmers. The large design space for the engine microarchitecture remains unexplored, and there is potential for new callbacks to unlock more applications. tākö provides the first step towards a polymorphic cache hierarchy, and we plan to explore each component further in future work.

ACKNOWLEDGMENTS

We thank the anonymous reviewers, Nikhil Agarwal, Souradip Ghosh, Graham Gobieski, Brandon Lucia, Sara McAllister, and Nathan Serafin for their feedback. Brian Schwedock is supported by an NSF Graduate Research Fellowship and the Ann and Martin McGuinn Graduate Fellowship. Jennifer Seibert was supported by an NSF REU grant in the REUSE program at CMU’s Institute for Software Research. This work was supported by NSF grant CCF-1845986.

REFERENCES

\cite{15} Nathan Beckmann and Daniel Sanchez. 2015. Jigsaw: Scalable Software-Defined Caches. In Proc. of the 22nd Intl. Conf. On Parallel Architectures and Compilation...