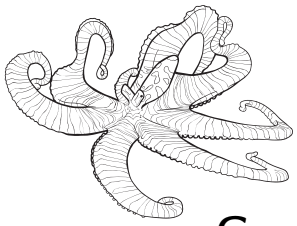


täkō: A Polymorphic Cache Hierarchy for General-Purpose Optimization of Data Movement

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:
Brian C. Schwedock, Piratach Yoovidhya, Jennifer Seibert, and Nathan Beckmann. 2022. täkō: A Polymorphic Cache Hierarchy for General-Purpose Optimization of Data Movement. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3470496.3527379

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.

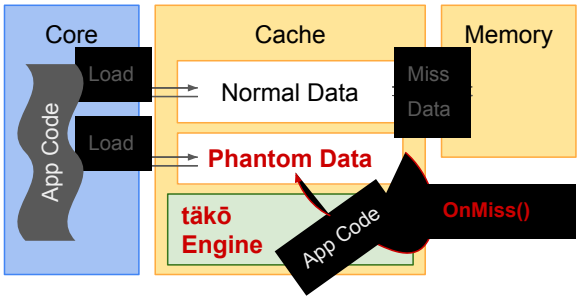


Figure 1: täkō in action: An application registers an address range whose semantics are defined by software callbacks. These callbacks run in-cache on programmable engines.

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

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06.
https://doi.org/10.1145/3470496.3527379

moves. We call this idea a *polymorphic cache hierarchy*, and we propose the *täkō*[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 täkō in action. Software (e.g., an application, domain-specific framework, or library) registers a *phantom address range* with täkō, 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], täkō adds programmable *engines* near caches to execute callbacks efficiently. In täkō, engines contain scheduling logic and a spatial dataflow fabric to run callbacks [43, 59, 103, 132, 138, 143]. With this microarchitectural support, täkō 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 täkō. This difference is the crux of the architecture: täkō 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. täkō'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 täkō, demonstrating that a general-purpose, programmable data-movement architecture can enable new functionality while approaching the performance of custom hardware.

- *In-cache data transformation:* täkō enables software-defined transformations (e.g., decompression) when data moves. With good locality, täkō eliminates redundant work to get 2.2× speedup and 61% energy savings.
- *Commutative scatter-updates:* täkō implements PHI [95], transforming the caches to use push-based semantics to accelerate commutative scatter-updates in graphs. täkō gets 4.2× speedup, similar to [95].
- *Decoupled graph traversals:* täkō implements HATS [92] as a representative decoupled streaming application. täkō accelerates graph traversals and gets a 43% speedup and 17% energy savings.
- *Transactions on non-volatile memory:* täkō's improved visibility over data movement eliminates wasteful work in NVM transactions. If no data is evicted before commit [91], täkō eliminates journaling overhead and achieves up to 2.1× speedup and 47% energy savings.
- *Detecting cache side-channel attacks:* täkō 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, täkō implements these applications on a single, general-purpose hardware design. täkō adds just ≈5% area overhead, similar to prior near-data systems. Further, we show that täkō's hardware achieves performance within 1.8% of an idealized design.

2 TÄKŌ OVERVIEW

täkō consists of software and hardware components. In software, täkō's programming interface gives software visibility and control over data movement via cache-triggered callbacks. In hardware, täkō 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. täkō 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 täkō's interface. Callbacks are registered only on selected addresses, and täkō 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 täkō system. On top of a baseline, cache-coherent multicore, each tile is augmented

[1] täkō is Japanese for octopus, an animal famous for its intelligence and mimicry. täkō is also a delicious Mexican-Asian fusion restaurant in Pittsburgh.

Table 1: täkō callback semantics.

Callback	Semantics	Side effects?*
`onMiss`	Generates data for requested address.	✗
`onEviction`	Handles eviction of unmodified data.	✗
`onWriteback`	Handles eviction of modified data.	✓

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

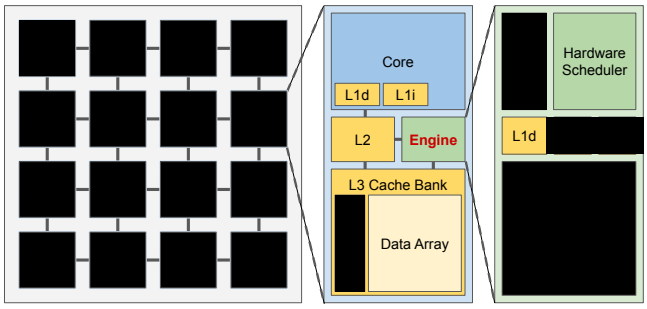


Figure 2: täkō adds programmable *engines* to each tile of a CMP. Engines schedule callbacks in response to cache events and execute them in parallel with conventional threads.

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 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.

```
int64 bases[N / 8]    # one base per line, or 8 values
int8 deltas[N]        # 4-bit exponent, 4-bit mantissa

total = 0

for idx in indices:
    # 1. decompress data
    base = bases[idx >> 3]
    delta = deltas[idx]
    mantissa = delta & 0b1111
    exponent = delta >> 4

    data = base + (mantissa << exponent)

    # 2. compute average
    total += data

avg = total / len(indices)
```

Figure 3: Example program written in traditional software.

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.

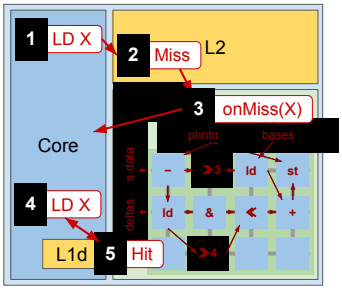
The modified program first registers the `Morph` at the private L2 cache, allocating a phantom address range for it. It then simply reads the decompressed data and computes the average, now using even simpler code. Fig. 5 illustrates its execution. The first time the program reads a phantom address `X`, 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

```
# define new cache semantics to pack data densely
class Decompressor extends täkō::Morph:
    int64* data  # actually in base Morph class
    int64* bases
    int8* deltas

    # decompress data when it moves into cache
    void onMiss(int64* phntmAddr):
        idx = phntmAddr - &this.data[0]

        base = bases[idx >> 3]
        delta = deltas[idx]
        mantissa = delta & 0b1111
        exponent = delta >> 4

        *phntmAddr = base + (mantissa << exponent)
```

```
int64 bases[N / 8]   # one base per line, or 8 values
int8 deltas[N]       # 4-bit exponent, 4-bit mantissa

# allocate new phantom address range w callbacks
morph = täkō::registerPhantom(
    Decompressor, PRIVATE, sizeof(int64) * N)
morph.bases = bases
morph.deltas = deltas

total = 0

# code now reads decompressed data directly
for idx in indices:
    total += morph.data[idx]

avg = total / len(indices)

morph.unregister()
```

Figure 4: Pseudocode for the same program in täkō.

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.



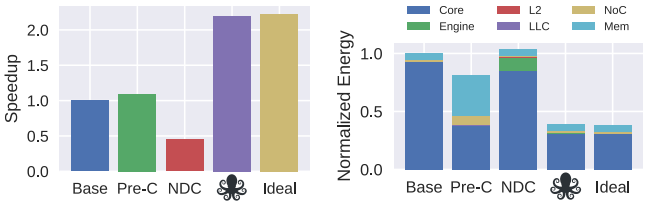


Figure 6: Results for the example program. täkō improves performance by 2.2× and reduces energy by 61%.

subsequent read of the same line (due to spatial or temporal locality) is a cache hit, *eliminating redundant work* from decompressing the same data many times.

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 (🐙) implementation. 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.)

täkō reduces execution time by 55% vs. the software baseline and by 50% vs. software pre-computation, and it reduces energy by 61% and 52%, respectively. Moreover, täkō comes within 1.1% performance and 1.3% energy of an idealized engine with unlimited, instantaneous, and energy-free compute.

täkō achieves these gains by memoizing decompressions of frequently accessed data (Fig. 7), greatly reducing the number of total decompressions. Although the pre-compute version avoids decompressing the same value multiple times, it decompresses values which are never accessed and also allocates memory for the entire decompressed array, incurring significant memory overheads. With täkō, decompression runs on in-cache engines, in parallel with software threads, similar to prior near-data computing (NDC) architectures. However, unlike NDC, *täkō triggers computation by data movement, not from cores:* instead of decompressing data every time it is requested, täkō decompresses data only on a miss and caches it thereafter, exploiting locality to eliminate redundant work [153, 154].

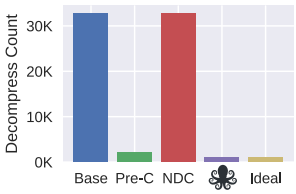


Figure 7: Num. decompressions.

This optimization is not possible in prior NDC systems, which move computation closer to data but do not improve software's visibility over data movement. Fig. 6 shows that NDC actually *hurts* performance and energy efficiency on this decompression example. This is because decompressing at the L2 fails to exploit locality in the L1s; in other words, offloading computation near-data is not always an optimization [83]. In contrast, täkō's *cache-triggered* computation gets the best of all worlds by executing computations near-data, eliminating wasteful work, and preserving locality.

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.

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.

```
1  Morph registerPhantom(morphType, location, size)
2  Morph registerReal(morphType, location, base, bound)
3  void flushData(morph)
4  void unregister(morph)
5
6  class Morph:
7      void* data              # base of address range
8      int size                # size of address range
9      Morph[] views           # engine-local state
10
11     # callbacks
12     void onMiss(addr)       # loads
13     void onEviction(addr)   # clean evictions
14     void onWriteback(addr)  # dirty evictions
```

Figure 8: täkō's interface for a polymorphic cache hierarchy.

4 TÄKŌ PROGRAMMING INTERFACE

täkō'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 täkō is available to application programmers, it currently targets experts; we envision täkō code being shipped as part of domain-specific frameworks or libraries.

Overview. täkō 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 `Morph`s 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 täkō'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 `täkō::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, täkō does not support `Morph`s at the L1 because L1s are very tightly integrated with cores; nor does it support `Morph`s 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 `Morph`s on existing data, `registerReal` accepts an arbitrary base and bound and attempts to register the `Morph` on this range. täkō 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 `unregister`ing 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` `Morph`s have a single view (at the L2), but `SHARED` `Morph`s 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 täkō's design. By defining callbacks in the `Morph`, software transforms the semantics of that address range. Callbacks are flexible to maximize täkō's applicability, but they must obey certain restrictions for correctness and performance.

Semantics. täkō 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. Intervening 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*

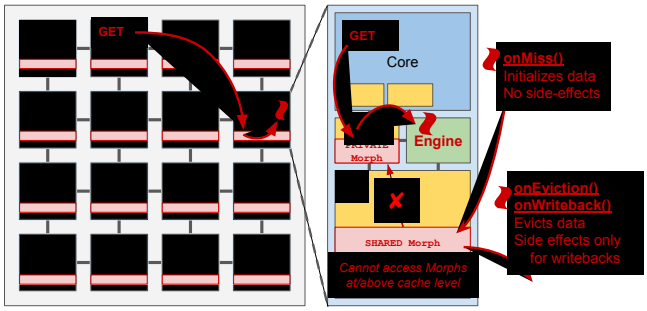


Figure 9: Callbacks are scheduled by hardware in response to cache misses, evictions, and writebacks, and run on the engine closest to the data. Only writebacks should have side effects.

be free of side effects,[2] 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 täkō 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. täkō 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 dataflow fabric when they first execute; in practice, one could compile code statically [130, 143].

Coherence and consistency. täkō 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.

[2]We define a side effect as a modification to non-local state, i.e., a store to any location except the engine's local `Morph` object or the `addr` itself.

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 täkō 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).

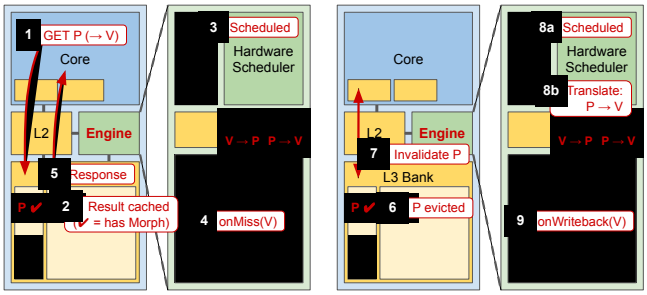
That said, more callbacks are certainly possible. `onReplacement` would allow software to optimize the eviction policy for particular workloads [10, 145]. `onHit` would allow customization of the cache coherence protocol, among other applications. We did not pursue `onHit` because programmable cache 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 — täkō has *no* performance impact on legacy applications.

5 TÄKŌ ARCHITECTURE

Similar to recent near-data-computing architectures [6, 83, 105, 142, 150], täkō 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 täkō.

Tracking `Morph`s. täkō 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 ❶. Alternatively, täkō could keep a separate table of registered `Morph`s, but this would limit the number of `Morph`s that could be registered concurrently.



(a) `onMiss`. (b) `onWriteback`.

Figure 10: Example callback execution in täkō on a phantom address. Engines schedule and execute callbacks in hardware. TLBs and cache tags track where `Morph`s are registered. The accompanying text walks through the steps of callback execution.

ISA. täkō 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 täkō.

State. Tags are extended with one bit to track whether a `Morph` is registered for the line at that cache level (2). 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 (3) (8a), which starts executing it as soon as the fabric is available and the callback configuration is loaded (4) (9). (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 (5). 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. täkō enforces this constraint by modifying its eviction policy, tr̃rîp (see below). For similar reasons, täkō 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, täkō modifies its RRIP-based [62] replacement policy, tr̃rîp, 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 $>4\times$.

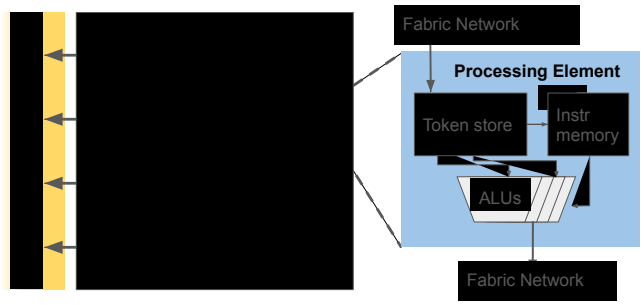


Figure 11: Sketch of engine dataflow fabric microarchitecture. A fabric of simple processing elements (PE) are connected by an on-chip network. Each PE holds a small number of instructions, which are issued to the ALU when input operands (with matching thread id) are available. A small number of PEs also connect to memory through the engine's L1 data cache.

5.3 Engine microarchitecture.

täkō 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 `Morph`s' 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., 5×5) 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 täkō engines because *(i)* callbacks are typically short, *(ii)* callbacks are frequently executed in parallel,

Table 2: Hardware overhead (state per L3 bank).

L3 tags	8K lines × 1 bit = 1 KB
Engine L1d, TLB, rTLB	8 KB + 2 KB + 2 KB = 12 KB
Callback buffer	8 lines × 64 B = 0.5 KB
Token store	25 PEs × 8 tokens / PE × 64 B = 12 KB
Instruction Memory	25 PEs × 16 instr / PE × ≈4 B = 1.6 KB
Total per L3 bank	27.1 KB / 512 KB = 5.3%

and *(iii)* callbacks are executed repeatedly. Short callbacks map easily onto a small, dynamic dataflow fabric, letting täkō run callbacks near-data with low area overhead. A dataflow fabric can easily run callbacks in parallel by assigning each a unique tag. Alternatively, täkō 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 täkō.

5.4 Putting it all together.

täkō'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, täkō 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 täkō 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 (8b). 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.

täkō 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' rTLB only needs to cover data currently in the cache, since `onEviction` and `onWriteback` 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 3: System parameters in our experimental evaluation.

Cores	16 cores, x86-64 ISA, 2.4 GHz, OOO Goldmont uarch [4]
Engines	16 engines, 15 int FUs (1-cycle latency), 10 mem FUs, 256-entry rTLB
L1	32 KB, 8-way set-assoc, split data and instruction caches
L2	128 KB, 8-way set-assoc, 2-cycle tag, 4-cycle data array, tree-PLRU repl., strided prefetcher
LLC	8 MB (512 KB per tile), 16-way set-assoc, 3-cycle tag, 5-cycle data array, inclusive, tree-PLRU repl.
NoC	mesh, 128-bit flits and links, 2/1-cycle router/link delay
Memory	4 controllers, 100-cycle latency, 11.8 GB/s per controller

OS support. täkō 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. `Morph`s 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 `Morph`s' data (i.e., using the `flushData` API). Doing this is feasible but takes time and energy, especially for `Morph`s at the `SHARED` cache.

Multi-tenancy, virtualization, and security. In heavily shared systems with many active `Morph`s, 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 täkō in execution-driven microarchitectural simulation. Our simulator shares infrastructure with SwarmSim [63], but supports cycle-level timing throughout the memory hierarchy and models täkō'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 täkō engine. Sec. 9 varies these parameters and shows that täkō 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 täkō has negligible impact on static power and to clearly distinguish täkō's impact on data movement energy from its overall performance benefits.

8 EVALUATION — CASE STUDIES ON TÄKŌ

täkō'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 täkō to demonstrate:

- täkō supports prior specialized cache hierarchies. We implement two prior designs that accelerate graphs in very different ways [92, 95].
- täkō enables features in software that are impossible without fine-grain visibility over data movement. Specifically, täkō lets the system eliminate unnecessary writes in direct-access NVM and detect suspicious activity.
- täkō'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. täkō 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 täkō 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 täkō 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 (1) miss in the cache, they trigger `onMiss` (2) 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 (3a) or appends them to a "bin" (3b), 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 täkō? PHI's design fits very well with täkō's interface. Its implementation requires application- and data-dependent operations on cache lines as they are allocated and evicted. This is exactly the type of data-movement control that täkō 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.

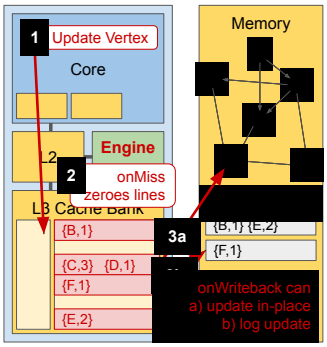


Figure 12: täkō lets software repurpose the cache to accelerate applications. PHI accelerates scatter-updates by buffering updates in-cache and applying them when evicted. Writebacks either apply updates in-place or log updates to be applied later. These optimizations are naturally implemented in täkō via `onMiss` and `onWriteback`.

Table 4: täkō callbacks for PHI.

Callback	Semantics
`onMiss`	Sets line to identity element (e.g., zero).
`onEviction`	—
`onWriteback`	If # updates > threshold, apply updates immediately; otherwise, log updates for application in "binning" phase.

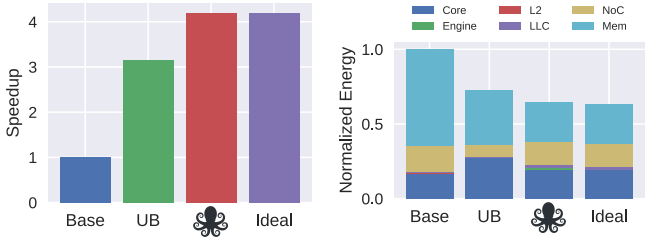


Figure 13: PHI results for PageRank on a 16M vertex, 160M edge synthetic graph. täkō improves performance by 4.2×.

Evaluation. Fig. 13 shows results for PageRank with 16 threads pushing updates to a single `Morph` registered at `SHARED`,[3] comparing täkō 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 täkō gets 4.2× speedup. täkō also reduces energy by 36%, compared to 27% for UB.

täkō 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. täkō reduces total accesses by 60% by buffering updates in-cache and only binning when

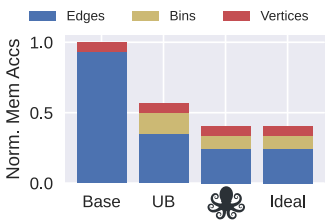


Figure 14: DRAM accs. per phase.

[3] Due to simulator limitations, we can currently only run PHI at a single level. But täkō'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, täkō 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 täkō 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 täkō'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 täkō 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 `onMiss`es 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. täkō 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 täkō? 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 täkō 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 täkō software lets it support a wide range of graph data formats or traversal heuristics [22, 94], unlike fixed-function hardware.

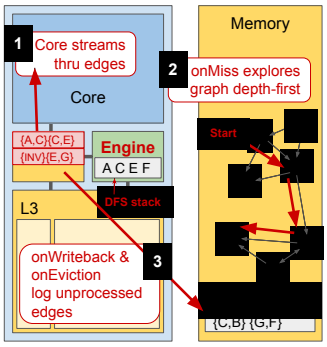


Figure 15: täkō's stream implementation supports complex decoupled pipelines. HATS improves locality in graphs by traversing the graph in bounded depth-first order so that communities are visited together. `onMiss` provides a simple, data-movement triggered approach to filling a stream.

Table 5: täkō callbacks for HATS.

Callback	Semantics
`onMiss`	Fills line with edges in BDFS order.
`onEviction`	Logs unprocessed edges.
`onWriteback`	Logs unprocessed edges.

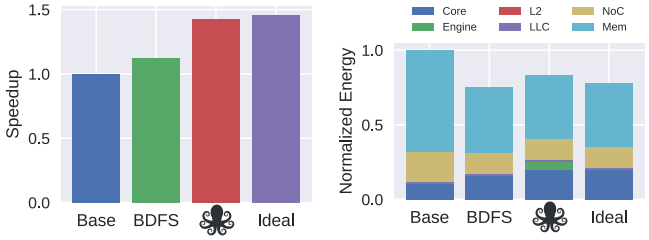


Figure 16: HATS results for one iteration of PageRank on uk-2002 graph [33]. täkō improves performance by 43% and reduces energy by 17% vs. the software baseline.

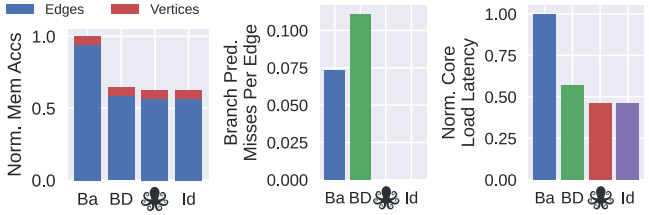


Figure 17: HATS performance breakdown. Left: DRAM accesses split by PageRank phase. Mid: core branch mispredictions per graph edge processed. Right: cumulative core load latency.

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, täkō, and an ideal engine. Baseline BDFS provides minimal benefits due to extra instructions with complex control flow. In contrast, täkō provides substantial speedup of 43%, approaching the 46% speedup of an ideal engine. täkō 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 täkō) reduces misses to vertex data during the edge

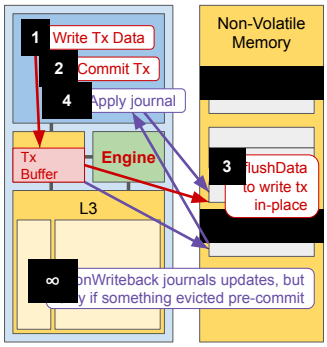


Figure 18: täkō increases software's visibility over data movement, making some operations dramatically more efficient. When caches are persistent, täkō lets transactions avoid journaling if there are no writebacks before commit. Better visibility thus greatly reduces transaction overheads.

phase by 40%. täkō also eliminates branch mispredictions by turning the complex BDFS traversal into simple loop over a sequential stream, whereas software BDFS increases mispredictions by 52%. Finally, täkō reduces memory latency seen at the core by 19% over BDFS by decoupling the edge traversal.

täkō 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 täkō. 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 (1). To commit a transaction, the thread simply flushes the `Morph`'s phantom data from the cache (2). `onWriteback` either writes directly to NVM (if the transaction has committed) or journals the writes (if not) (∞). In the common case where no data is evicted, täkō adds minimal overhead (3). But if data is evicted before commit, then the application must apply the journaled writes to commit the transaction (4). 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 täkō? 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. täkō 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.

Table 6: täkō callbacks for NVM support.

Callback	Semantics
`onMiss`	Sets line with INVALID value.
`onEviction`	—
`onWriteback`	If transaction committed, apply writes immediately; otherwise, journal writes.

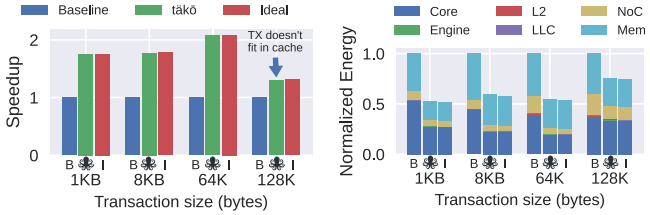


Figure 19: Results for NVM journaling microbenchmark at different transaction sizes. täkō improves performance by up to 2.1× and reduces energy by up to 47%.

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, täkō provides up to 2.1× speedup by eliminating unnecessary journaling. täkō executes ≈50% fewer core instructions and ≈36% fewer total instructions (Fig. 20), yielding large speedup and up to 47% energy savings. täkō 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, täkō still outperforms the baseline by filling the journal in `onWriteback`, off the critical-path of the core.

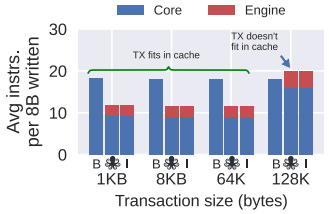


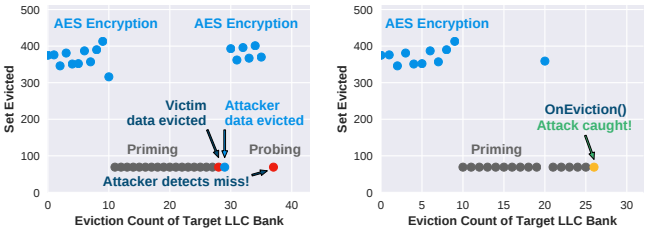
Figure 20: Instructions executed for each 8B written in application.

8.4 Detecting side-channel attacks.

Finally, we demonstrate täkō'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 täkō 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 sets the victim



(a) Attack succeeds in baseline. (b) Attack detected in täkō.

Figure 21: Prime+probe attack on AES encryption tables at the L3. Without täkō, the attack succeeds with the victim unaware. täkō detects the attack immediately.

Table 7: täkō callbacks for detecting side-channel attacks.

Callback	Semantics
`onMiss`	—
`onEviction`	Interrupt main thread.
`onWriteback`	—

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 in-order core and ideal. Dataflow vastly outperforms in-order, 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).

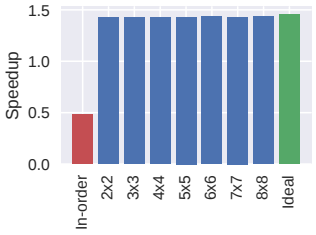


Figure 22: Sensitivity to engine fabric with HATS.

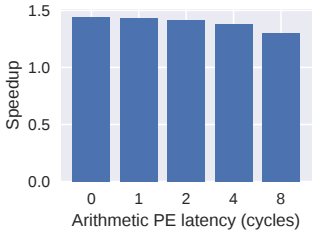


Figure 23: Sensitivity to arithmetic PE latency with HATS.

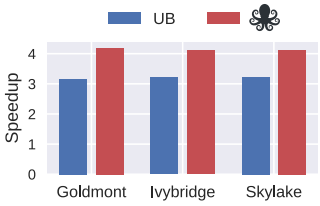


Figure 24: PHI results for PageRank on a 160M-edge synthetic graph. täkō performs similarly with all core microarchitectures.

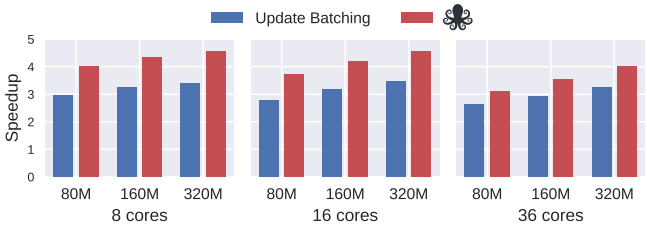


Figure 25: PHI results for PageRank across different numbers of cores/threads and different graph sizes (shown as num. edges). täkō performs well across all configurations.

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 `onWriteback`s 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 [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 [32, 38], and a significant fraction of programs will always run on general-purpose cores [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 [2, 50], graph analytics [92, 95, 150], data structures [54, 58, 154], memoization [8, 40, 153, 154], compression [9, 36, 90, 106, 107, 118, 136, 146], data layout [7, 23, 155], prefetching [6, 131, 149], coherence and synchronization [34, 75, 151, 152], memory management [85, 135], and system software [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 [15, 29, 37, 39, 110, 117, 133], software policies [16, 17, 24, 66, 82, 120], or a richer interface [93, 137]. These works are complementary to täkō: they control data movement behind the load-store interface, whereas täkō 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 [27, 44, 64, 71, 72, 80, 98, 101, 104, 124, 128] or near high-bandwidth memory [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 [5, 57, 83, 134, 148]. Other architectures enable near-data computing within a CPU's memory hierarchy, letting cores offload work to memory [5, 51] or caches [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. täkō 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 [3, 23, 56, 73, 113, 123]. More recently, designs have added some programmability to the memory hierarchy for specific purposes: e.g., prefetching [6, 131] or compression [146]. By contrast, täkō 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. täkō 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 täkō 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. täkō 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

[1] Sarita V Adve and Kourosh Gharachorloo. 1996. Shared memory consistency models: A tutorial. *IEEE Computer* (1996).
[2] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2017. Compute caches. In *Proc. of the 23rd IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-23).*
[3] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. 1995. The MIT Alewife machine: architecture and performance. *Proc. of the 22nd annual Intl. Symp. on Computer Architecture* (1995).
[4] Agner Fog. 2020. The microarchitecture of Intel, AMD and VIA CPUs. https://www.agner.org/optimize/microarchitecture.pdf.
[5] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *Proc. of the 42nd annual Intl. Symp. on Computer Architecture (Proc. ISCA-42).*
[6] Sam Ainsworth and Timothy M Jones. 2018. An Event-Triggered Programmable Prefetcher for Irregular Workloads. In *Proc. of the 23rd intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXIII).*
[7] Berkin Akin, Franz Franchetti, and James C Hoe. 2015. Data reorganization in memory using 3D-stacked DRAM. In *Proc. of the 42nd annual Intl. Symp. on Computer Architecture (Proc. ISCA-42).*
[8] Ismail Akturk and Ulya R Karpuzcu. 2017. AMNESIAC: Amnesic Automatic Computer. (2017).
[9] Alaa R Alameldeen and David A Wood. 2004. Adaptive cache compression for high-performance processors. In *Proc. of the 31st annual Intl. Symp. on Computer Architecture (Proc. ISCA-31).*
[10] Vignesh Balaji, Neal Crago, Aamer Jaleel, and Brandon Lucia. 2021. P-OPT: Practical Optimal Cache Replacement for Graph Analytics. In *Proc. of the 27th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-27).*
[11] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. 2014. Near-data processing: Insights from a MICRO-46 workshop. *IEEE Micro* 34, 4 (2014), 36–42.
[12] Sahan Bandara and Michel A Kinsy. 2020. Adaptive caches as a defense mechanism against cache side-channel attacks. *Journal of Cryptographic Engineering* (2020).
[13] Scott Beamer, Krste Asanovic, and David Patterson. 2015. Locality exists in graph processing: Workload characterization on an Ivy Bridge server. In *Proc. of the IEEE Intl. Symp. on Workload Characterization (Proc. IISWC).*
[14] Scott Beamer, Krste Asanović, and David Patterson. 2017. Reducing PageRank communication via propagation blocking. In *Proc. of the 31st IEEE Intl. Parallel and Distributed Processing Symp. (Proc. IPDPS).*
[15] Nathan Beckmann and Daniel Sanchez. 2013. Jigsaw: Scalable Software-Defined Caches. In *Proc. of the 22nd intl. conf. on Parallel Architectures and Compilation*

Techniques.

[16] Nathan Beckmann and Daniel Sanchez. 2015. Talus: A Simple Way to Remove Cliffs in Cache Performance. In *Proc. of the 21st IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-21).*

[17] Nathan Beckmann, Po-An Tsai, and Daniel Sanchez. 2015. Scaling distributed cache hierarchies through computation and data co-scheduling. In *Proc. of the 21st IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-21).*

[18] Bryan Black. 2013. Die Stacking is Happening!. In *MICRO-46 Keynote.*

[19] Guy E Blelloch, Phillip B Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. 2018. The parallel persistent memory model. In *Proc. of the 30th ACM Symp. on Parallelism in Algorithms and Architectures (Proc. SPAA).*

[20] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. 2018. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *Proc. of the 23rd intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXIII).*

[21] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. 1999. Web caching and Zipf-like distributions: Evidence and implications. In *IEEE INFOCOM.* 126–134.

[22] Aydin Buluç and John R. Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *Proc. of the 22nd IEEE Intl. Parallel and Distributed Processing Symp. (Proc. IPDPS).*

[23] John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, Lambert Schaelicke, and Terry Tateyama. 1999. Impulse: Building a smarter memory controller. In *Proc. of the 5th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-5).*

[24] Shuang Chen, Christina Delimitrou, and José F. Martinez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proc. of the 24th intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXIV).*

[25] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B Gibbons, Todd C Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. 2008. Flexible hardware acceleration for instruction-grain program monitoring. In *Proc. of the 35th annual Intl. Symp. on Computer Architecture (Proc. ISCA-35).*

[26] Sanchuan Chen, Fangfei Liu, Zeyu Mi, Yinqian Zhang, Ruby B. Lee, Haibo Chen, and XiaoFeng Wang. 2018. Leveraging Hardware Transactional Memory for Cache Side-Channel Defenses. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security.*

[27] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-43).*

[28] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proc. of the ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (Proc. OOPSLA).*

[29] Intel corporation. 2015. Improving Real-Time Performance by Using Cache Allocation Technology. *Intel Whitepaper* (2015).

[30] William J. Dally. 2010. GPU Computing: To Exascale and Beyond. In *Supercomputing '10, Plenary Talk.*

[31] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J-H Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. 2003. Merrimac: Supercomputing with streams. In *Proc. of the ACM/IEEE conf. on Supercomputing (Proc. SC03).*

[32] William J. Dally, Yatish Turakhia, and Song Han. 2020. Domain-Specific Hardware Accelerators. *Commun. ACM* 63, 7 (June 2020), 10 pages. https://doi.org/10.1145/3361682

[33] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM TOMS* 38, 1 (2011).

[34] Christopher De Sa, Matthew Feldman, Christopher Ré, and Kunle Olukotun. 2017. Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent. In *Proc. of the 44th annual Intl. Symp. on Computer Architecture (Proc. ISCA-44).*

[35] Joao Mario Domingos, Nuno Neves, Nuno Roma, and Pedro Tomás. 2021. Unlimited Vector Extension with Data Streaming Support. In *Proc. of the 48th annual Intl. Symp. on Computer Architecture (Proc. ISCA-48).*

[36] Magnus Ekman and Per Stenstrom. 2005. A robust main-memory compression scheme. In *Proc. of the 32nd annual Intl. Symp. on Computer Architecture (Proc. ISCA-32).*

[37] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sanchez. 2018. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores. In *Proc. of the 24th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-24).*

[38] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger. 2011. Dark Silicon and The End of Multicore Scaling. In *Proc. of the 38th annual Intl. Symp. on Computer Architecture (Proc. ISCA-38).*

[39] Yaosheng Fu, Tri M Nguyen, and David Wentzlaff. 2015. Coherence domain restriction on large scale systems. In *Proc. of the 48th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-48).*

[40] Adi Fuchs and David Wentzlaff. 2018. Scaling Datacenter Accelerators With Compute-Reuse Architectures. In *Proc. of the 45th annual Intl. Symp. on Computer Architecture (Proc. ISCA-45).*

[41] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. 2015. Practical near-data processing for in-memory analytics frameworks. In *Proc. of the 24th Intl. Conf. on Parallel Architectures and Compilation Techniques (Proc. PACT-24).*

[42] Mingyu Gao and Christos Kozyrakis. 2016. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *Proc. of the 22nd IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-22).*

[43] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. 2021. Snafu: An Ultra-Low-Power, Energy-Minimal CGRA-Generation Framework and Architecture. In *Proc. of the 48th annual Intl. Symp. on Computer Architecture (Proc. ISCA-48).*

[44] Maya Gokhale, Bill Holmes, and Ken Iobst. 1995. Processing in memory: The Terasys massively parallel PIM array. *Computer* 28, 4 (1995).

[45] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R Reed Taylor, and Ronald Laufer. 1999. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proc. of the 26th annual Intl. Symp. on Computer Architecture (Proc. ISCA-26).*

[46] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection Using Hardware Transactional Memory. In *Proceedings of the 26th USENIX Conference on Security Symposium.*

[47] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptol. ePrint Arch.* 2016 (2016), 204.

[48] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache games–Bringing access-based cache attacks on AES to practice. In *2011 IEEE Symposium on Security and Privacy.*

[49] Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry. 1992. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Scalable shared memory multiprocessors.* Springer, 167–192.

[50] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pdream, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-43).*

[51] Milad Hashemi, Eiman Ebrahimi, Onur Mutlu, Yale N Patt, et al. 2016. Accelerating dependent cache misses with an enhanced memory controller. In *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-43).*

[52] Zecheng He and Ruby B Lee. 2017. How secure is your cache against side-channel attacks?. In *Proc. of the 50th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-50).*

[53] John Hennessy and David Patterson. 2018. A New Golden Age for Computer Architecture: Domain-Specific Hardware/Software Co-Design, Enhanced Security, Open Instruction Sets, and Agile Chip Development. In *Turing Award Lecture.*

[54] Byungchul Hong, Gwangsun Kim, Jung Ho Ahn, Yongkee Kwon, Hongsik Kim, and John Kim. 2016. Accelerating linked-list traversal through near-data processing. In *Proc. of the 25th Intl. Conf. on Parallel Architectures and Compilation Techniques (Proc. PACT-25).*

[55] Mark Horowitz. 2014. Computing's energy problem (and what we can do about it). In *ISSCC.*

[56] Mark Horowitz, Margaret Martonosi, Todd C Mowry, and Michael D Smith. 1996. Informing memory operations: Providing memory performance feedback in modern processors. (1996).

[57] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. 2016. Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems. In *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-43).*

[58] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. 2016. Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In *Proc. of the 34th Intl. Conf. on Computer Design (Proc. ICCD).*

[59] Yuanjie Huang, Paolo Ienne, Olivier Temam, Yunji Chen, and Chengyong Wu. 2013. Elastic cgras. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays (FPGA).*

[60] Intel. 2020. Intel Optane Persistent Memory 200.

[61] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES. In *2015 IEEE Symposium on Security and Privacy.*

[62] Aamer Jaleel, Kevin Theobald, Simon C. Steely Jr, and Joel Emer. 2010. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *Proc. of the 37th annual Intl. Symp. on Computer Architecture.*

[63] Mark C Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A scalable architecture for ordered parallelism. In *Proc. of the 48th annual*

IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-48).

[64] Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Lam, Pratap Pattnaik, , and Josep Torrellas. 1999. FlexRAM: Towards an intelligent memory system. In *Proc. of the 17th Intl. Conf. on Computer Design (Proc. ICCD).*

[65] David Kaplan, Jeremy Powell, and Tom Woller. 2016. *AMD Memory Encryption.* Technical Report. AMD.

[66] Harshad Kasture and Daniel Sanchez. 2014. Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads. In *Proc. of the 19th intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XIX).*

[67] Rajat Kateja, Nathan Beckmann, and Gregory R Ganger. 2020. Tvarak: software-managed hardware offload for redundancy in direct-access NVM storage. In *Proc. of the 47th annual Intl. Symp. on Computer Architecture (Proc. ISCA-47).*

[68] M. Kayaalp, D. Ponomarev, N. Abu-Ghazaleh, and A. Jaleel. 2016. A high-resolution side-channel attack on last-level cache. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC).* 1–6. https://doi.org/10.1145/2897937.2897962

[69] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. 2001. Imagine: media processing with streams. *IEEE Micro* 21, 2 (2001).

[70] Vladimir Kiriansky, Yunming Zhang, and Saman Amarasinghe. 2016. Optimizing indirect memory references with milk. In *Proc. of the 25th Intl. Conf. on Parallel Architectures and Compilation Techniques (Proc. PACT-25).*

[71] Peter M Kogge. 1994. EXECUBE-A new architecture for scaleable MPPs. In *Proc. of the intl conf. on Parallel Processing (ICPP).*

[72] Christoforos E Kozyrakis, Stylianos Perissakis, David Patterson, Thomas Anderson, Krste Asanovic, Neal Cardwell, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, Randi Thomas, Noah Treuhaft, and Katherine Yelick. 1997. Scalable processors in the billion-transistor era: IRAM. *Computer* 30, 9 (1997).

[73] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, , and John Hennessy. 1994. The Stanford FLASH multiprocessor. In *Proc. of the 21st annual Intl. Symp. on Computer Architecture (Proc. ISCA-21).*

[74] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens.. In *Proc. of the 12th USENIX symp. on Operating Systems Design and Implementation (Proc. OSDI-12).*

[75] Joo Hwan Lee, Jaewoong Sim, and Hyesoon Kim. 2015. BSSync: Processing near memory for machine learning workloads with bounded staleness consistency models. In *Proc. of the 24th Intl. Conf. on Parallel Architectures and Compilation Techniques (Proc. PACT-24).*

[76] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. 2020. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science* 368, 6495 (2020).

[77] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1990. The directory-based cache coherence protocol for the DASH multiprocessor. (1990).

[78] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2008. Statistical properties of community structure in large social and information networks. In *Proc. of the intl. World Wide Web conf. (WWW-17).*

[79] Bo Li and Bo Jiang. 2018. Cache Attack on AES for Android Smartphone. In *Proceedings of the 2nd International Conference on Cryptography, Security and Privacy.*

[80] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. 2016. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Proceedings of the 53rd Annual Design Automation Conference.* ACM, 173.

[81] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy.*

[82] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: improving resource efficiency at scale. In *Proc. of the 42nd annual Intl. Symp. on Computer Architecture (Proc. ISCA-42).*

[83] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, and Nathan Beckmann. 2020. Livia: Data-centric computing throughout the memory hierarchy. In *Proc. of the 25th intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXV).*

[84] Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (Proc. PLDI).*

[85] Martin Maas, Krste Asanovic, and John Kubiatowicz. 2018. A Hardware Accelerator for Tracing Garbage Collection. In *Proc. of the 45th annual Intl. Symp. on Computer Architecture (Proc. ISCA-45).*

[86] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution without Checkpoints. In *Proc. of the ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (Proc. OOPSLA).*

[87] Kiwan Maeng and Brandon Lucia. 2018. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *Proc. of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18).* USENIX Association, Berkeley, CA, USA, 16 pages. http://dl.acm.org/citation.cfm?id=3291168.3291178

[88] Milo Martin, Mark D Hill, and Daniel J Sorin. 2012. Why on-chip cache coherence is here to stay. *Commun. ACM* (2012).

[89] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. 2015. DoppelgäNger: A Cache for Approximate Computing. In *Proc. of the 48th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-48).*

[90] Joshua San Miguel and Natalie Enright Jerger. 2016. The anytime automaton. In *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-43).*

[91] Kevin E Moore, Jayaram Bobba, Michelle J Moravan, Mark D Hill, and David A Wood. 2006. LogTM: log-based transactional memory.. In *Proc. HPCA.*

[92] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In *Proc. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-51).*

[93] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2016. Whirlpool: Improving dynamic cache management with static data classification. In *Proc. of the 21st intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXI).*

[94] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2017. Cache-Guided Scheduling: Exploiting Caches to Maximize Locality in Graph Processing. In *1st International Workshop on Architectures for Graph Processing (AGP 2017), held in conjuntion with ISCA-44.*

[95] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2019. PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates. In *Proc. of the 52nd annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-52).*

[96] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Rao Naveed Bin Rais, Vianney Lapotre, and Guy Gogniat. 2018. Run-time detection of prime+ probe side-channel attack on AES encryption algorithm. In *2018 Global Information Infrastructure and Networking Symposium (GIIS).*

[97] Quan M. Nguyen and Daniel Sánchez. 2021. Fifer: Practical Acceleration of Irregular Applications on Reconfigurable Architectures. In *Proc. of the 54th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-54).*

[98] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. 1993. The J-machine multicomputer: an architectural evaluation. In *Proc. of the 20th annual Intl. Symp. on Computer Architecture.*

[99] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. In *ISCA 44.*

[100] Hamza Omar and Omer Khan. 2020. IRONHIDE:A Secure Multicore that Efficiently Mitigates Microarchitecture State Attacks for Interactive Applications. In *Proc. of the 26th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-26).*

[101] M. Oskin, F. Chong, and T. Sherwood. 1998. Active Pages: A Model of Computation for Intelligent Memory. In *Proc. of the 25th annual Intl. Symp. on Computer Architecture (Proc. ISCA-25).*

[102] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers' Track at the RSA Conference.* Springer, 1–20.

[103] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. 2013. Triggered instructions: A control paradigm for spatially-programmed architectures. In *Proc. of the 40th annual Intl. Symp. on Computer Architecture (Proc. ISCA-40).*

[104] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. 1997. A case for intelligent RAM. *IEEE micro* 17, 2 (1997), 34–44.

[105] Ashutosh Pattnaik, Xulong Tang, Onur Kayiran, Adwait Jog, Asit Mishra, Mahmut T Kandemir, Anand Sivasubramaniam, and Chita R Das. 2019. Opportunistic computing in gpu architectures. In *Proc. of the 46th annual Intl. Symp. on Computer Architecture (Proc. ISCA-46).*

[106] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. 2013. Linearly compressed pages: a low-complexity, low-latency main memory compression framework. In *Proc. of the 46th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-46).*

[107] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. 2012. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proc. of the 21st Intl. Conf. on Parallel Architectures and Compilation Techniques (Proc. PACT-21).*

[108] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. 2020. Optimus prime:

Accelerating data transformation in servers. In *Proc. of the 25th intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXV).*

[109] Seth H Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, and Vijayalakshmi Srinivasan. 2014. NDC: Analyzing the Impact of 3D-Stacked Memory + Logic Devices on MapReduce Workloads. In *Proc. of the IEEE Intl. Symp. on Performance Analysis of Systems and Software (ISPASS).*

[110] M.K. Qureshi and Y.N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. of the 39th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-39).*

[111] Moinuddin K Qureshi. 2018. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *Proc. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-51).*

[112] Moinuddin K Qureshi. 2019. New attacks and defense for encrypted-address cache. In *Proc. of the 46th annual Intl. Symp. on Computer Architecture (Proc. ISCA-46).*

[113] Steven K Reinhardt, James R Larus, and David A Wood. 1994. Tempest and Typhoon: User-level shared memory. In *Proc. of the 21st annual Intl. Symp. on Computer Architecture (Proc. ISCA-21).*

[114] Thomas J Repetti, João P Cerqueira, Martha A Kim, and Mingoo Seok. 2017. Pipelining a triggered processing element. In *Proc. of the 50th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-50).*

[115] Gururaj Saileshwar, Prashant J Nair, Prakash Ramrakhyani, Wendy Elsasser, Jose A Joao, and Moinuddin K Qureshi. 2018. Morphable counters: Enabling compact integrity trees for low-overhead secure memories. In *Proc. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-51).*

[116] Daniel Sanchez and Christos Kozyrakis. 2010. The ZCache: Decoupling Ways and Associativity. In *Proc. of the 43rd intl. symp. on Microarchitecture.*

[117] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *Proc. of the 38th annual Intl. Symp. on Computer Architecture (Proc. ISCA-38).*

[118] Somayeh Sardashti and David A Wood. 2013. Decoupled compressed cache: exploiting spatial locality for energy-optimized compressed caching. In *Proc. of the 46th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-46).*

[119] Mahadev Satyanarayanan, Nathan Beckmann, Grace A. Lewis, and Brandon Lucia. 2021. The Role of Edge Offload for Hardware-Accelerated Mobile Devices. In *HotMobile.*

[120] Brian C. Schwedock and Nathan Beckmann. 2020. Jumanji: The Case for Dynamic NUCA in the Datacenter. In *Proc. of the 53rd annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-53).*

[121] André Seznec. 1993. A case for two-way skewed-associative caches. In *Proc. of the 20th annual Intl. Symp. on Computer Architecture.*

[122] André Seznec. 1994. Decoupled sectored caches: conciliating low tag implementation cost. In *Proc. of the 21st annual Intl. Symp. on Computer Architecture (Proc. ISCA-21).*

[123] Ofer Shacham, Zain Asgar, Han Chen, Amin Firoozshahian, Rehan Hameed, Christos Kozyrakis, Wajahat Qadeer, Stephen Richardson, Alex Solomatnikov, Don Stark, Megan Wachs, and Mark Horowitz. 2009. Smart memories polymorphic chip multiprocessor. In *Proc. of the 46th Design Automation Conf. (Proc. DAC-46).*

[124] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. (2016).

[125] Chaoqun Shen, Congcong Chen, and Jiliang Zhang. 2021. Micro-architectural cache side-channel attacks and countermeasures. In *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC).*

[126] Matthew D Sinclair, Johnathan Alsop, and Sarita V Adve. 2017. Chasing away rats: Semantics and evaluation for relaxed atomics on heterogeneous systems. In *Proc. of the 44th annual Intl. Symp. on Computer Architecture (Proc. ISCA-44).*

[127] Dimitrios Skarlatos, Nam Sung Kim, and Josep Torrellas. 2017. Pageforge: a near-memory content-aware page-merging architecture. In *Proc. of the 50th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-50).*

[128] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating graph processing using ReRAM. In *Proc. of the 24th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-24).*

[129] Arun Subramaniyan, Jingcheng Wang, Ezhil RM Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. 2017. Cache automaton. In *Proc. of the 50th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-50).*

[130] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. 2003. WaveScalar. In *Proc. of the 36th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-36).*

[131] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, et al. 2021. Prodigy: Improving the Memory Latency of Data-Indirect Irregular Workloads Using Hardware-Software Co-Design. In *Proc. of the 27th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-27).*

[132] Christopher Torng, Peitian Pan, Yanghui Ou, Cheng Tan, and Christopher Batten. 2021. Ultra-Elastic CGRAs for Irregular Loop Specialization. In *Proc. of the 27th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-27).*

[133] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Jenga: Software-Defined Cache Hierarchies. In *Proc. of the 44th annual Intl. Symp. on Computer Architecture (Proc. ISCA-44).*

[134] Po-An Tsai, Changping Chen, and Daniel Sanchez. 2018. Adaptive Scheduling for Systems with Asymmetric Memory Hierarchies. In *Proc. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-51).*

[135] Po-An Tsai, Yee Ling Gan, and Daniel Sanchez. 2018. Rethinking the Memory Hierarchy for Modern Languages. In *Proc. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-51).*

[136] Po-An Tsai and Daniel Sanchez. 2019. Compress objects, not cache lines: An object-based compressed memory hierarchy. In *Proc. of the 24th intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXIV).*

[137] Nandita Vijaykumar, Abhilasha Jain, Diptesh Majumdar, Kevin Hsieh, Gennady Pekhimenko, Eiman Ebrahimi, Nastaran Hajinazar, Phillip B Gibbons, and Onur Mutlu. 2018. A case for richer cross-layer abstractions: Bridging the semantic gap with expressive memory. In *Proc. of the 45th annual Intl. Symp. on Computer Architecture (Proc. ISCA-45).*

[138] Dani Voitsechov and Yoav Etsion. 2014. Single-graph multiple flows: Energy efficient design alternative for GPGPUs. (2014).

[139] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. 1997. Baring it all to software: Raw machines. *IEEE Computer* 30, 9 (1997).

[140] Zhengrong Wang and Tony Nowatzki. 2019. Stream-based memory access specialization for general purpose processors. In *Proc. of the 46th annual Intl. Symp. on Computer Architecture (Proc. ISCA-46).*

[141] Zhengrong Wang, Jian Weng, Sihao Liu, and Tony Nowatzki. 2022. Near-Stream Computing: General and Transparent Near-Cache Acceleration. (2022).

[142] Zhengrong Wang, Jian Weng, Jason Lowe-Power, Jayesh Gaur, and Tony Nowatzki. 2021. Stream Floating: Enabling Proactive and Decentralized Cache Optimizations. In *Proc. of the 27th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-27).*

[143] Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu, and Tony Nowatzki. 2020. A hybrid systolic-dataflow architecture for inductive matrix algorithms. In *Proc. of the 26th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-26).*

[144] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proc. of the 26th Symp. on Operating System Principles (Proc. SOSP-26).*

[145] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. 2017. Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks. In *Proc. of the 44th annual Intl. Symp. on Computer Architecture (Proc. ISCA-44).*

[146] Yifan Yang, Joel S Emer, and Daniel Sanchez. 2021. SpZip: Architectural Support for Effective Data Compression In Irregular Applications. In *Proc. of the 48th annual Intl. Symp. on Computer Architecture (Proc. ISCA-48).*

[147] Younis A. Younis, Kashif Kifayat, and Abir Hussain. 2017. Preventing and Detecting Cache Side-Channel Attacks in Cloud Computing. In *Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing.*

[148] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-oriented Programmable Processing in Memory. In *Proc. HPDC.*

[149] Dan Zhang, Xiaoyu Ma, and Derek Chiou. 2016. Worklist-directed Prefetching. *IEEE Computer Architecture Letters* (2016).

[150] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. 2018. Minnow: Lightweight Offload Engines for Worklist Management and Worklist-Directed Prefetching. In *Proc. of the 23rd intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXIII).*

[151] Guowei Zhang, Virginia Chiu, and Daniel Sanchez. 2016. Exploiting Semantic Commutativity in Hardware Speculation. In *Proc. of the 49th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-49).*

[152] Guowei Zhang, Webb Horn, and Daniel Sanchez. 2015. Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems. In *Proc. of the 48th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-48).*

[153] Guowei Zhang and Daniel Sanchez. 2018. Leveraging Hardware Caches for Memoization. *Computer Architecture Letters (CAL)* 17, 1 (2018).

[154] Guowei Zhang and Daniel Sanchez. 2019. Leveraging caches to accelerate hash tables and memoization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture.* 440–452.

[155] Jialiang Zhang, Michael Swift, and Jing (Jane) Li. 2022. Software-Defined Address Mapping: A Case on 3D Memory. In *Proc. of the 27th intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc.*

ASPLOS-XXVII).

[156] Lu Zhang and Steven Swanson. 2019. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *Proc. of the USENIX Annual Technical Conf. (Proc. USENIX ATC)*.

[157] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing communication for PIM-based graph processing with efficient data partition. In *Proc. of the 24th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-24)*.