METAL: Caching Multi-level Indexes in Domain-Specific Architectures

Anagha M. Anil Kumar[*]
ama241@sfu.ca
Simon Fraser University
Burnaby, BC, Canada

Aditya Prasanna[*]
apa120@sfu.ca
Simon Fraser University
Burnaby, BC, Canada

Jonathan Balkind
jbalkind@ucsb.edu
UC Santa Barbara
Santa Barbara, CA, USA

Arrvindh Shriraman
ashriram@sfu.ca
Simon Fraser University
Burnaby, BC, Canada

Abstract

State-of-the-art domain specific architectures (DSAs) work with sparse data, and need hardware support for index data-structures [31, 43, 57, 61]. Indexes are more space-efficient for sparse-data, and reduce DRAM bandwidth, if data reuse can be managed. However, indexes exhibit dynamic accesses, chase pointers, and need to walk-and-search. This inflates the working set and thrashes the cache. We observe that the cache organization itself is responsible for this behavior.

We develop METAL, a portable caching idiom that enables DSAs to employ index data-structures. *METAL* decouples reuse of the index metadata from data reuse, and optimizes it independently. We propose two ideas: i) *IX-Cache:* A cache that leverages range tags to short-circuits index walks, and reduces the working set. *IX-cache* helps capture the trade-off between wider index nodes that maximize reach vs those that are closer to leaf and minimize walk latency. ii) *Reuse Patterns:* An interface to explicitly manage the cache. Patterns orchestrate cache insertions and bypass as we dynamically traverse different index regions. *METAL* improves performance vs. streaming DSAs by 7.8×, address-caches by 4.1×, and state-of-the-art DSA-cache [50] by 2.4×. We reduce DRAM energy by 1.6× vs. prior state-of-the-art.

CCS Concepts: • **Computer systems organization** → **Architectures**; • **Hardware** → **Hardware-software codesign**.

Keywords: Domain-specific Architectures, Caches, Dataflow architectures, Indexes

[*]Both authors contributed equally to the paper

ACM Reference Format:

Anagha M. Anil Kumar, Aditya Prasanna, Jonathan Balkind, and Arrvindh Shriraman. 2024. METAL: Caching Multi-level Indexes in Domain-Specific Architectures. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3620665.3640402

1 Introduction

Limitations in technology scaling has led to a shift away from general-purpose architectures to more specialized domain-specific architectures (DSAs). Modern DSAs [14, 43, 49, 57, 60, 61] transform high-level programs into pipelined dataflow graphs, utilizing specialized computation and memory blocks. To maximize memory bandwidth and space utilization, currently DSAs need to work with a variety of index data-structures such as compressed tensors [10, 27, 36, 39], trees [22], hash-tables and sets [45, 46]. Unfortunately, DSAs are not equipped to capture index reuse and only support streaming algorithms [14, 24, 57]. Some sparse GEMM DSAs [16, 60, 61] do capture reuse, but hardwire the cache for GEMM, and are not portable. Further, prior DSA caches [13, 50] only target leaf reuse, and exhibit poor hit rate due to repeated walks.

Address-based caches are a well-understood idiom for capturing reuse in dynamic data structures [4, 31]. However, they have organizational flaws. We elaborate using a textbook index, the B+tree (Fig.1:Left). Each node includes a set of sorted keys, along with pointers to a sub-range at the next level. There are multiple challenges: i) **Challenge 1: Address-caches always require root-to-leaf walks, which overflow cache capacity.** In an address-cache, tag lookups use addresses. Since each node's address is only available from its parent, walks always traverse the whole B+tree. Even if a lower node is cached (e.g., 11-21), we cannot reach it until the parent is read. **Challenge 2: Address-caches are polluted with redundant nodes.** To minimize latency, the address-cache must hold all nodes along a walk (since all are touched). However, we could kickstart from level-3 if we could identify it in the cache without walking the redundant level-2 and level-1. **Challenge 3: Tradeoff between reach and effectiveness.** Upper nodes are common across walks

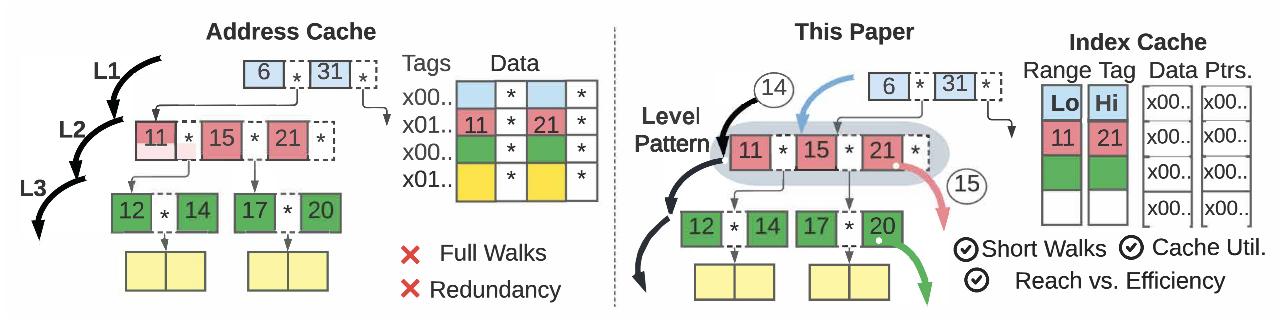


Figure 1. B+tree Illustration. Address Cache vs. Index (IX) Cache

and maximize reach, while lower nodes effectively short-circuit. The choice is DSA dependent e.g., SpMM prefers short-circuiting, while JOIN prefers reach.

METAL introduces two complementary ideas to address the challenges of caching index data-structures: i) *IX-cache* (Index Cache), a novel cache architecture that uses key ranges as cache tags. It short-circuits index walks and reduces the working set. ii) *Reuse patterns* to manage cache insertion and bypass in *IX-cache*. Reuse patterns observe that index or key space serves as an affine abstraction for expressing caching preferences. Fig.1:Right illustrates. The *IX-cache* inverts the organization of an address-cache, and the [Lo, Hi] range in the index node constitutes the tag (previously the data-block in the address-cache). The DSA probes the *IX-cache* using index keys, and on a hit, can kickstart the walk from the cached node closer to the leaf. Fig.1:Right, key (15) starts the walk from cached [11-21]. Interestingly, beyond reducing walk latency, the primary benefit of this short-circuiting is a reduced working set. Per-tag matches require more energy (than address-cache), but fewer accesses mean lower overall energy.

According to the algorithm's reuse pattern, *IX-cache* can cache upper-level wider-range indexes that can short-circuit more walks, or cache lower-level indexes that short-circuit more effectively and minimize walk latency. While walks are dynamic and chase pointers, patterns express caching and bypassing on affine index features (e.g., ranges, levels). The index features are mapped to block addresses at runtime. Fig.1 shows the level pattern which targets level 2 in the B+Tree. As we walk the index to (14), we only insert the level-2 node, [11-21], into *IX-cache* and bypass other green nodes. This frees up space to cache red entries along other branches (e.g., right branch of 31) to improve reach. We identify three generalized reuse patterns exhibited and exploitable across multiple DSAs, applications, and index types.

We incorporate *METAL* into four DSAs: *Gorgon* [56], *Capstan* [49], *Aurochs* [57], and *Widx* [31] and target multiple algorithms. In § 2.1 we provide an overview of the DSAs we target and in § 3 we provide a detailed description. We evaluate index reuse in multiple applications: data analytics, database scans, graph processing, spatial analysis, and sparse matrix algebra. Our contributions:

- We propose *METAL* an architectural template for enabling DSAs to work with multi-level indexed data-structures such as trees, hash tables, and fibers.
- We create a novel cache architecture, *IX-cache*, that uses index ranges as cache tags. Reorganizing the cache helps short-circuit index walks and improves caching efficiency.
- We identify three reuse patterns and descriptors which enable *METAL* to manage the tradeoff between *IX-cache*'s reach and effectiveness. Patterns express caching policies on index or key space, which is affine, instead of non-affine addresses.
- *METAL* performs 4.1× better than address-cache, and 2.4× better than X-cache [50] (state-of-the-art DSA cache). We save 1.7× bandwidth vs. address and 1.6× vs. X-cache.

2 Motivation and Scope

In this section, we first preview the types of architectures we target. We then summarize the common traits of index data-structures that need to be considered during caching. Finally, we discuss why existing cache organizations do not suffice.

2.1 Indexed Data-Structures in Target DSAs

Here we provide an overview of our target DSAs (Fig. 2) and focus on interaction with data structures. The DSAs we study are organized similarly: the computation is laid out in a grid of compute tiles [14, 15, 31, 37, 49, 56, 57, 57, 60] (Figure 2). They vary in the parallelism within each tile: some exploit task parallelism [57], others exploit vector parallelism [56], and loop parallelism [14]. Each system targets a different memory pattern, and type of index. Gorgon [56] supports declarative patterns (e.g., map, filter) on relational data that scan through ranges of records. The index is a table of records, and the primary reuse is the mid-level roots. Widx [31] supports lookups and joins on relational data that perform nearest neighbor scans. Widx predates DSAs and continues to rely on address-caches. *Capstan* [49] targets sparse tensor algebra with matrices represented as fibres. Fibres are indexes in which coordinates of the non-zeroes constitute the keys. *METAL* enables Capstan to work with dynamic tensors [10]

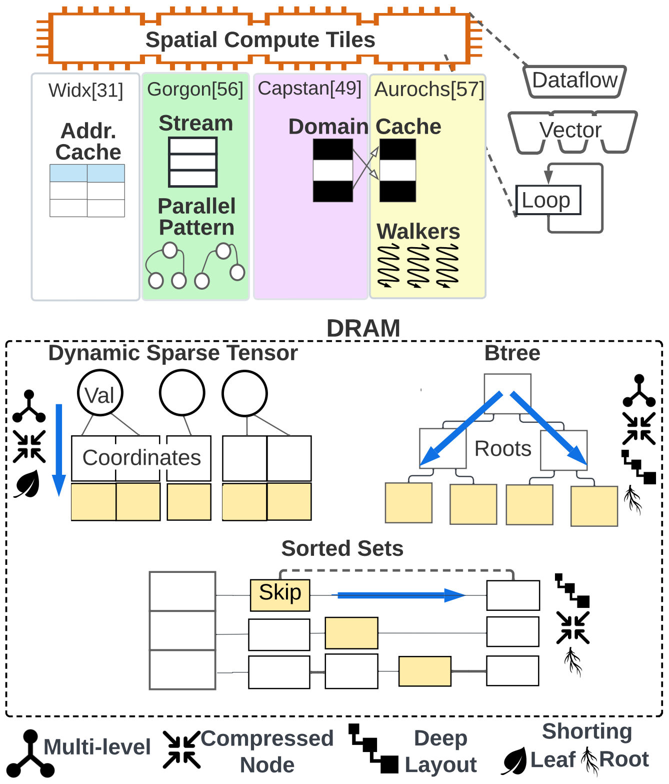


Figure 2. Indexes used in domain-specific architectures. Blue arrows indicate walk pattern.

and supports leaf-level scans. *Aurochs* [57] scans through the records in an unordered manner; *METAL* speeds up these unordered scans. We elaborate in more detail in § 4.

In the DSAs we study, 30-90% of end-to-end time is from walks. The DSAs walk index data-structures to access data objects [50, 57]. In sparse algebra, key-value stores, and database joins where we scan through the index, walks are 70-80% of total time. For data analytics with heavy computational work, index walks are 30-40% of total time. In workloads that employ nested data-dependent index types (e.g., RTree, Database clauses) the walks are 50% of total time.

2.2 Common Traits of Widely-used Index Types

In our paper, we study multiple index types currently in use across different fields: B+trees (from databases [56]), sorted sets (from Redis [45]), R-Trees (from spatial [49]), and dynamic sparse tensors (from sparse-matrix algebra [10]). At first glance, these data-structures seem uniquely tailored for their specific applications. However, we show that they share several common traits from a hardware standpoint. i) **Hierarchical structure with internal roots:** Whether viewed horizontally (hash indexes[1]) or vertically (tree-like indexes), the data-structures are organized hierarchically into layers with internal-roots. These roots enable the skipping of large parts of the data structure. Thus, it is crucial that hardware caches can quickly identify these roots. ii) **Single key lookup and Short-Circuit potential:** Indexes, while organized across multiple levels, use the same key for checks at each internal root i.e., the lookup at every root can be redundant. This provides the opportunity to 'short-circuit' a traversal by caching any internal root. In contrast, data-structures such as tries and prefix-trees uses a portion of the key at each internal root, and we need to cache all roots along a path. iii) **Compressed internal roots:** Each internal root only allocates metadata for keys actually in use, and maintains them compactly as ranges. The metadata could be coordinates in sparse tensors or ranges in a B+Tree. Overall, this allows us to cache the actual node itself, as opposed to just a pointer to the node like in bitmap tries [5, 59]. v) **Deep layouts:** We find that indexes are typically deep, and include multiple levels of pointer indirection. In our research, we have studied B+trees up to 18 levels deep. Such depth is beneficial for sparse key spaces, as it creates balanced searches, compacts internal roots, and improves storage efficiency. However, this also means the cache has to focus on the inherent tradeoff between caching common roots (and maximizing reach) versus targeting specific branches (and maximizing short-circuiting). v) **Range scans and ordered traversals:** We find that both range scans[30, 38] (i.e., maps and iterators) and point queries (i.e., lookups) are common depending on domain. Prior work in hardware focused predominantly on point queries [50]. Many index structures maintain some form of order - either keys (like in B-trees) or spatial (like in R-trees). We can use this ordering to understand the access pattern and cache the roots.

2.3 *METAL* vs. other caches

Observation 1: *In address caches, walks repeatedly traverse index and thrash the cache due to inflated working set.*
Observation 2: *DSA caches [50] cache leaves and short-circuit on hits, but have high miss rate since leaves have minimal reuse in deep index data-structures.*
Observation 3: *METAL caches intermediate nodes with high reuse (maximizing reach), and short-circuits walks.*

Fig. 3 highlights the benefits of *METAL*'s cache organization over address-cache and X-cache [50] (the state-of-the-art DSA cache). **Walking (or Working) region** shades the nodes touched during an index walk i.e., the working set. **Caching Region** shades index nodes kept in the cache during a walk (helps reuse). The hit and miss paths vary between cache designs. In an address-cache, hit or miss, each walk traverses all levels from root-to-leaf. Different searches touch different branches, and the walking region encompasses the entire index. A hit eliminates only a single DRAM access. X-cache [50] tags the data with the actual key, and a hit short-circuits the entire walk. However, on a miss, X-Cache triggers

[1]Our target is hash tables with chaining that exhibit hierarchical accesses; not open chaining.

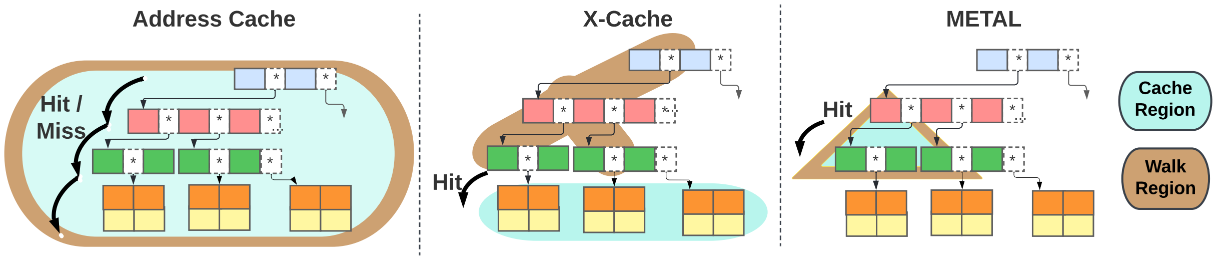


Figure 3. Work (or Walk) Region Vs Cache Region in Address Cache, X-Cache and *METAL*.

Table 1. METAL vs. state-of-the-art storage idioms (shaded cells indicate limitations)

		Scratch+DMA Plasticine [42],Buffets [40]	**Access-Exe.** Aurochs [57],SJoin [14]	**Address-Cache** MAD [25], Widx [31]	**DSA Cache** X-Cache [50]	**METAL**
Target	Algorithm	Perfect Loops	Streaming	Any	Any	Cache-conscious
	LD/ST order	Limited (on-chip only)	Only FIFO	Arbitrary	Flexible	
	Data structure	Tensors and Affine	Streams	Any	Key-Value	Index Data-Structures
Design	Short-circuit	No. Walking always required.			Fixed (Leaf)	Flexible (Any)
	Bandwidth	Low	High (threads)	High(working set)	High (misses) .	Low (flexible)
	Temporal reuse	Yes (credit-driven)	No reuse	Yes (LRU)	Leaf	Indexes
	Spatial reuse	Yes (tiles)	No reuse	Yes (Blocks)	Leaf	Indexes
	Reuse Orch.	Explicit	N/A	Implicit	Implicit	Patterns
	Utilization	High (credit)		Low (overflows)	Low (misses)	Flexible.

a root-to-leaf walk. It targets the wrong reuse as the leaf levels are least reused, tend to be large, and overflow the cache. *METAL* can choose the index nodes and levels to cache to manage the working set, and control the tradeoff between reach and effectiveness. i) *METAL* can cache upper nodes that can cooperatively short-circuit across multiple branches to increase the effective hit rate. X-cache cannot target reach at all, and thus has high miss rate. ii) *METAL* can also cache lower nodes close to leaves and reduce walk latency. For the interested reader we perform a detailed exposition on each of the effects discussed here in §5.1. Table 1 qualitatively compares *METAL* with the state of the art. *METAL* helps DSAs work with index data-structures that exhibit dynamic and non-affine access patterns.

3 *METAL* Architecture

Fig. 4 shows *METAL* incorporated into a spatial dataflow architecture [42]. Spatial architectures map the computation to a grid of the compute tiles. Each tile implements a dataflow thread [57]; a vessel that encapsulates the user-specified function along with register state sufficient to run the thread. Our LLVM compiler places the operations on the grid of functional units [47, 58] using high-level-synthesis. Physically, the compute tiles are connected to a 2.5D high-bandwidth-memory (HBM) via an interposer (similar to current GPUs). A DMA engine interface shuttles data between the DRAM and on-chip storage.

Logically, the compute tiles interface with the data-structure using keys (not addresses). Each data object in the index has a unique key which provides a namespace, that loads and stores can use. The objects in DRAM are fetched by walkers, state-machines that traverse the data-structure and chase pointers [50, 57]. Hardware handles the address calculations and cache operations to get the data. The data object itself is allocated in a separate region in the DRAM and are accessed through a DMA interface or stream (depends on the particular DSA design). The index only contains the pointers to the data object, and our cache only targets the index traversal itself. Each tile includes a local scratchpad for staging the leaf data objects and capturing immediate reuse of fields within the object; it also acts as a defacto write buffer.

There is also a global scratchpad for preloading and double buffering from the DRAM. *METAL* adds two components: i) an *IX-cache* shared by multiple compute tiles to maximize cooperative caching, ii) a pattern controller that directs cache block traffic between the compute tiles, walker and *IX-cache*.

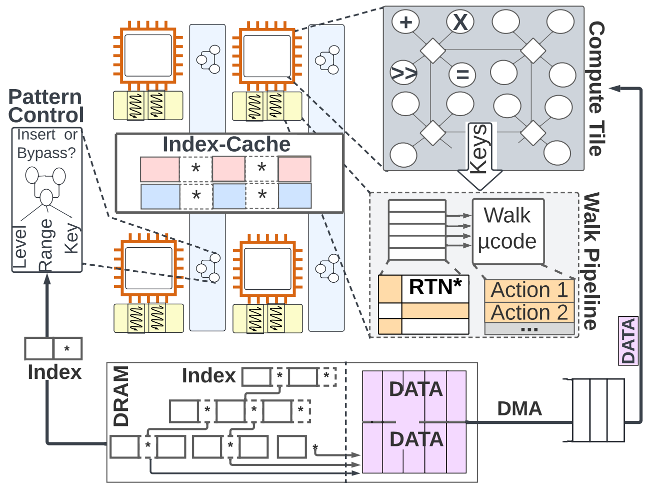
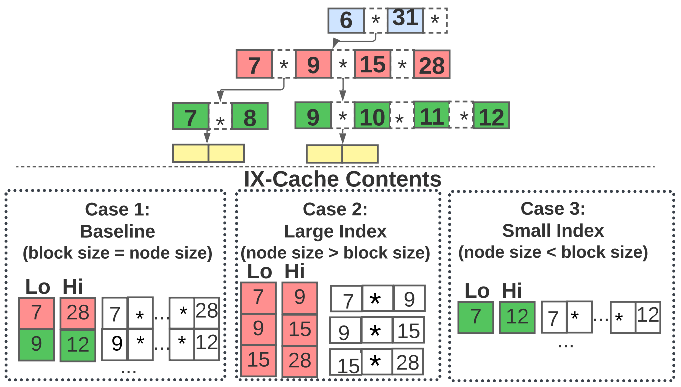


Figure 4. Overview of *METAL* Microarchitecture

The controller is simply a state machine (it does not require complex logic or SRAM).

3.1 Index Cache (*IX-cache*)

Cache Block. The *IX-cache*'s block includes child keys and pointers from an index node. The block is tagged with the [Lo,Hi] tuple, which represent the smallest and largest keys (the range) stored in the block. Fig. 5 a) shows the possible layouts: i) Case 1: When `block size == node size`, the cache tag [Lo,Hi] stores the exact range. e.g., we tag the red block with [Lo=7,Hi=28], the node's end keys. ii) Case 2: When `node size > block size`, the cache block holds a sub-range. Here the node [7-28] is split into three entries, [7-9],[9-15], and [15-28]. Each entry holds one of the child pointers. c) Case 3: When `node size < block size`, the cache coalesces multiple nodes in the same level and stores a super-range e.g., here the cache block fuses the two nodes [7-8], and [9-12].



IX-Cache Contents
Case 1: Baseline (block size = node size)
Case 2: Large Index (node size > block size)
Case 3: Small Index (node size < block size)
Lo Hi

Figure 5. Packing index nodes into cache blocks

Hit Path. The *IX-cache* is designed to short-circuit walks. Specifically, when the *IX-cache* identifies a hit, it provides a pointer that facilitates the walk's continuation much closer to the desired leaf, and in some scenarios, directly pointing to the leaf. Every block in *IX-cache* tagged with the [Lo,Hi] tuple which represents key ranges encompassed by the block. Fig. 6 illustrates the stages in the pipeline (only the first is required, and remaining are optional): i) Matching stage: We use the range tags to match with the incoming key and check entries for $Lo \leq key \leq Hi$. An exact match bypasses the remaining step e.g., in Fig. 6 key $\textcircled{7}$ will match [7-15]). Like address, the tags are maintained in SRAM. They are read-out to registers with comparators attached. ii) *Prioritize ties:* In instances where multiple matches arise, a 'level field' helps break the tie by deciding the match to prioritize. For illustration, a key marked $\textcircled{10}$ might match both the red and green ranges in the cache. However, priority would be given to [9-11]. A bitmap aids in maintaining relative priority. iii) Finally, we read the cached index node from the data array and extract the next child pointer. Each cache block (which represents an index node) includes a set of sorted keys along with child pointers. We find the child to be followed based on where the key falls in the set of keys e.g., here $\textcircled{10}$ will match 9-10 in the block [9*10*11]. We achieve this with: parallel $\leq$ across all the index keys, then find first bit from the right (first $>$). In cases where the node does not fit in the block, we split the node ranges across multiple blocks.

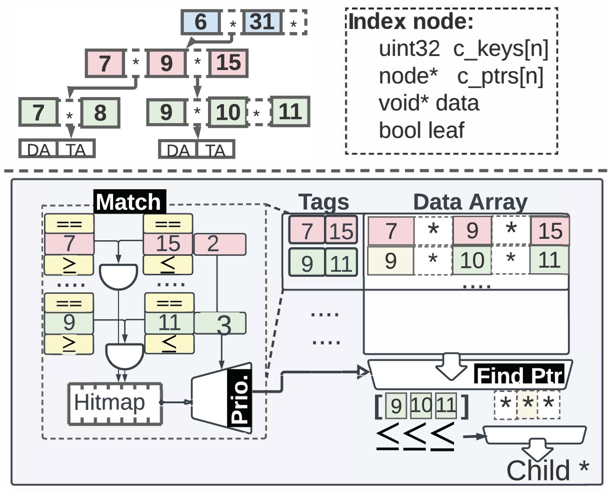
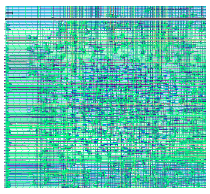


Figure 6. Hit Path: *IX-cache* Logic



Comparator Logic.						
Ref.	nm	Vdd	Trans	Bit.	mW	ns
[11, 55]	180	1.8	800	64	0.7	0.5
[41]	90	1.0	1051	64	1	0.23
[7]	90	1.2	—	64	0.9	0.85
[19]	90	1.0	1359	64	0.8	0.22
This Paper	45	0.85	1400	2×32	0.02	1

Figure 7. Right: Synthesis of tag match logic in Nangate 45nm. depth = 10, entries = 256. Left: Floorplan

We implement a segmented tag match in Chisel (other tag implementations are possible [55]) and compare our complexity against prior literature (Fig. 7). As an upper-bound estimate, we synthesize using nangate 45nm PDK and OpenROAD [3].

Set-Associativity. *IX-cache* can be made set-associative similar to an address-cache. Like an address, index key values are divided into blocks (of size 2^b) and sets. Block bits (b) come from the LSB of the key. The keys are logically divided into 16 (b = 4) wide blocks e.g., keys 0-to-15 will be a block (Fig 8). Every index node will be mapped to the same set as the keys it contains. Thus, index node [11-15] and [12-13] will map to set 0 (they are $\in 0-15$). There are a few differences: i) the key space is virtual i.e., no physical backing memory. Thus, block size impacts the position of the set bits, but not spatial locality. ii) Each index node only includes a sub-range of the key block e.g., here [11-15] does

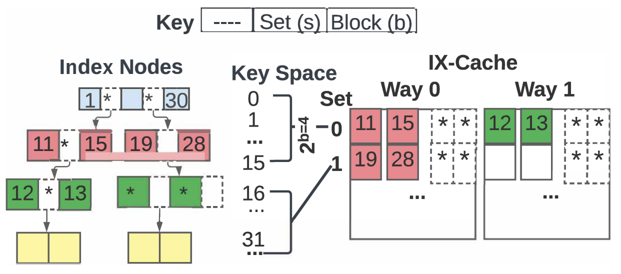


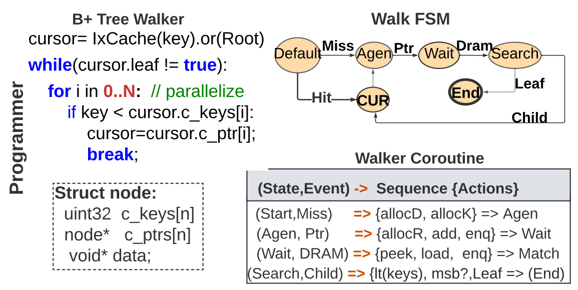
Figure 8. Set-associative IX-Cache

not include [0-10]. This is why we use range tags. iii) We are caching an index of the key space, not the key space itself. Larger block sizes can exacerbate set conflicts as more nodes may map to the same set e.g., In Fig. 8, if the block size was 32 (5 bits), nodes [11-15], [12-13], and [19-28] would all map to set 0, leading to conflicts that limit capacity.

3.2 Walk Pipeline

Miss Path: A miss triggers a root-to-leaf walk. *METAL* repurposes the prior microcode engines that the DSAs already include [50, 57]. We only provide an overview due to lack of space. Fig. 9 shows the index node, pseudo code, FSM, and microcode table. The walk itself is highly serial and data-dependent since key values determine the next child pointer. However, each walker refills the data independently. The goal is to harvest memory-level parallelism from these independent walks. For this, we break the walker into a set of states. At each long-latency state we yield to other requests. In the hardware pipeline we multiplex multiple walks on a single thread. There are two yield points in this index example: i) Wait: accessing the current cursor, refilled from DRAM. ii) Search: searching the node's internal keys to find the next pointer. The steps are compiled to a table and microcode.

As the walker traverses the index, the pattern controller directs the insertion policy for the *IX-cache*. The controller's policy is explicitly set based on latent behavior and reuse patterns exhibited by the applications (described in the next section). For any node during a walk, the descriptor determines whether a specific node should be inserted into the *IX-cache* or bypassed entirely. The decision is based on index node information such as the level and range provided by the walker and matches the directive of the pattern.



B+ Tree Walker

```
cursor= IxCache(key).or(Root)
while(cursor.leaf != true):
  for i in 0..N:  // parallelize
    if key < cursor.c_keys[i]:
      cursor=cursor.c_ptr[i];
      break;
```

```
Struct node:
 uint32  c_keys[n]
 node*   c_ptrs[n]
 void* data;
```

Walker Coroutine

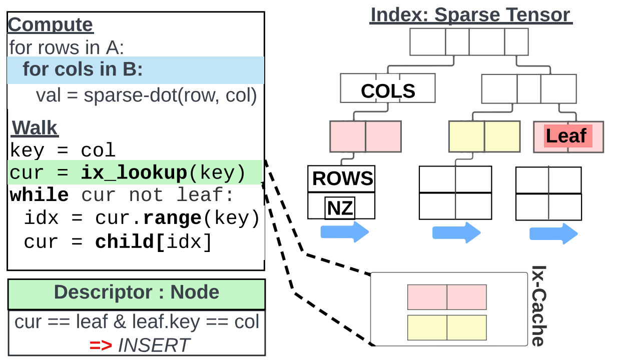
(State,Event) -> Sequence {Actions}
(Start,Miss) => {allocD, allocK} => Agen
(Agen, Ptr) => {allocR, add, enq} => Wait
(Wait, DRAM) => {peek, load, enq} => Match
(Search,Child) => {lt(keys), msb?,Leaf => (End)

Figure 9. Cache miss handler.

4 Reuse Patterns and Descriptors

In this section, we motivate the need for a pattern-based approach in reasoning about locality in index data-structures. Traditionally, we determine locality (both temporal and spatial) based on the stream of addresses, since they reflect the DRAM layout. However, in DSAs, the keys used to access the data do not reflect the data-structure layout. index data-structures are self-balancing and organize keys for lookups rather than an application's scan behavior pattern. We will now describe the foundational concepts: i) **Reuse patterns (blue arrows in figures below):** Our insight is that if we observed the collective behavior of walks, we can identify cooperative reuse of index nodes across them. More precisely, we define a reuse pattern as the minimal set of nodes touched by an "ideal" walker to capture a group of application keys, achieving maximal short-circuiting and maximal reach. ii) **Cache descriptors (green bars):** Cache descriptors are a pragma or hint that *METAL* uses to express reuse patterns to the *IX-cache*. The cache descriptors explicitly indicate what type of index nodes should be cached. More importantly, it refrains from specifying the exact node which permits the cache to adjust to the exact pointers chased by the walks. Compared to the "ideal" walker of a reuse pattern, practical *METAL* is constrained by *IX-cache*'s size and geometry. Descriptors help manage the tradeoff between competing goals of short-circuiting (effectiveness) and reach. Our investigation of DSAs leads us to identify three reuse patterns that generalize applications, and we exploit them using descriptors.

4.1 Reuse Pattern: Node. Target: SpMM [49]



Compute

```
for rows in A:
  for cols in B:
    val = sparse-dot(row, col)
```

Walk

```
key = col
cur = ix_lookup(key)
while cur not leaf:
  idx = cur.range(key)
  cur = child[idx]
```

Descriptor : Node

cur == leaf & leaf.key == col
=> *INSERT*

Figure 10. Illustration of Node Reuse. **Target:** Sparse Matrix Multiplication. A×B; only B shown for clarity.

Fig. 10 illustrates the reuse pattern in SpMM. The data is laid out in state-of-the-art dynamic sparse tensors [10, 39] (in § 5.2 we also evaluate fibers [35]). We only show matrix B for clarity. The non-zero (NZ) column ids are indexed in a B+Tree; the leaves hold the NZs and their rows ids. The computation is an inner product that uses a series of sparse dot products. The reuse pattern is along the leaves of the matrix as the inner loop tries to retrieve NZs of matrix B whose coordinates match. The blue lines show the reuse pattern along the leaves as we retrieve NZs from each col of B (which maps to a leaf).

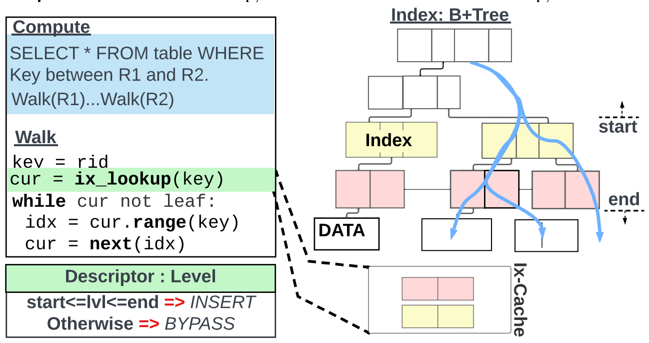
To capture this behavior we introduce the node descriptor. The descriptor indicates to the underlying *IX-cache* that it should target nodes at the leaves only, bypassing other nodes entirely. Now walks will effectively be completely short-circuited when the walk hits in the *IX-cache*. We remove all indirect accesses in the common case. For this pattern we also lock down the entry in the cache for a specific duration e.g., here we set life based on the number of accesses to block. In SpMM, life is set to the number of non-zeros in each column. The descriptor can target any node in the index; but here SpMM demanded leaves.

4.2 Reuse Pattern: Level. Target: Database Scan [56]

Range scans are queries of the following type: *SELECT * FROM table WHERE X BETWEEN R1 AND R2* that retrieve all records in [R1,R2]. Fig. 11 illustrates the computation and walk pattern. As the walkers repetitively traverse the index from root-to-leaf, they start exhibiting level-wise reuse. The walks funnel through common nodes at intermediate levels, but then diverge as they approach their destination leaves. We use a level descriptor to capture this reuse pattern. It has two goals i) cache nodes that are common across walks to maximize reach, ii) bypass nodes uncommon across walks and save cache space. In the figure we are set to gather nodes between orange and red; and walks can now exploit the common nodes that are in *IX-cache* to short-circuit. Levels from root-to-orange are not cached since they are redundant, and levels below red are bypassed since they are uncommon. Level descriptor adjusts the start and end levels based on the utility. If the utility is low, the band is adjusted to maximize reach [start-δ, end]. If the level utility is high, the band is expanded [start, end+δ] to improve short-circuiting effectiveness. The descriptor tracks the utility of each level as $\frac{\#Total-Access}{\#Nodes-touched}$.

4.3 Reuse Pattern: Level, Branch. Target: Spatial Analysis [26]

RTree is a spatial data structure used primarily for indexing multi-dimensional information. Each object in the RTree corresponds to a bounding box that contains all the geometric

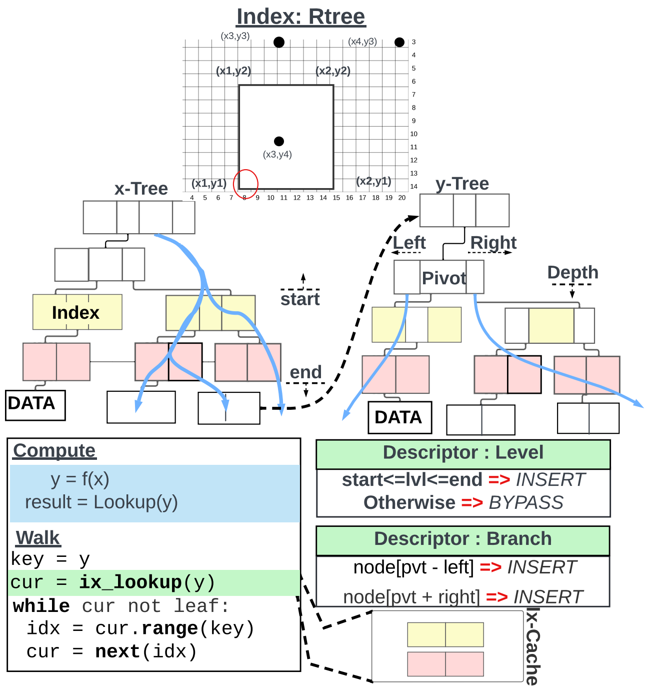


```
Compute
SELECT * FROM table WHERE
Key between R1 and R2.
Walk(R1)...Walk(R2)

Walk
key = rid
cur = ix_lookup(key)
while cur not leaf:
  idx = cur.range(key)
  cur = next(idx)
```

Descriptor : Level
start<=lvl<=end => *INSERT*
Otherwise => *BYPASS*

Figure 11. Illustration of Level Reuse. **Target:** Range Scans



```
Compute
    y = f(x)
  result = Lookup(y)

Walk
key = y
cur = ix_lookup(y)
while cur not leaf:
  idx = cur.range(key)
  cur = next(idx)
```

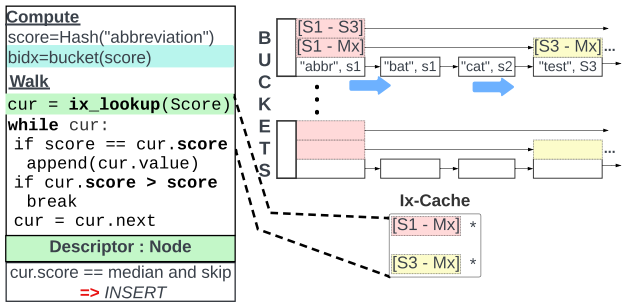
Descriptor : Level
start<=lvl<=end => *INSERT*
Otherwise => *BYPASS*

Descriptor : Branch
node[pvt - left] => *INSERT*
node[pvt + right] => *INSERT*

Figure 12. Illustration of Branch Descriptor. **Target:** Spatial Analysis. Tree-y shown for demonstration.

objects represented by vertices. In fig. 12, we demonstrate the application space using quadrilateral embedding. We implement a two-dimensional RTree which contains quadrilaterals bound by x and y coordinates; each of the coordinates are indexed in a BTree with the leaf values in the x-tree serving as keys to the y-tree, as it has information of the neighbouring y-coordinates that form a quadrilateral. The algorithm tries to find quadrilaterals within a certain bound.

We first generate random x co-ordinates and traverse the x-tree. Once we reach the leaf, we get the y-tree keys that correlate to these x keys to form quadrilaterals. The scan of the y tree is narrowed to specific x coordinates found, we find a pattern of certain key clusters being repetitively scanned i.e., the reuse tends to be along certain tree sub-branches. We use the branch cache descriptor to capture this reuse pattern. The branch descriptor adjusts both depth and breadth based on the size of the cluster of keys. We use the median of the key cluster to serve as a pivot (Fig. 12), and proceed to cache sub-branches to the left and right up to a certain depth. The descriptor improves the hit rate for keys that can potentially form quadrilaterals. Branch descriptor adjusts the pivot, left, right and depth as the key cluster changes; for this we maintain a moving average for a window of keys. If the utility is low and the cache is not full, the features are adjusted to maximize sub-tree reuse. If the hit rate in the sub-region is maintained and there is space in the *IX-cache*, we increase the depth, to improve short-circuiting.



```
score=Hash("abbreviation")
bidx=bucket(score)
```

```
cur = ix_lookup(Score)
while cur:
 if score == cur.score
  append(cur.value)
 if cur.score > score
  break
 cur = cur.next
```

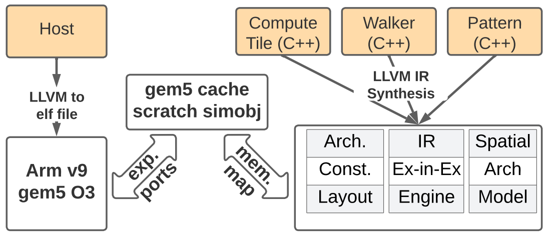
cur.score == median and skip
=> *INSERT*

Figure 13. Illustration of sorted sets. **Target:** Node reuse

4.4 Reuse Pattern: Node. Target: Sorted Sets [45, 46]

We implement sorted sets (from Redis[45, 46, 62]), a type of hash index commonly used in tagging [52], categorization, and auto-completion [2]. This is a hybrid data structure combining characteristics of sets and hashes. Each record in the sorted set is a tuple: a string and a number referred to as the score. Nodes are mapped to buckets in a hash-table based on their score. Each bucket is implemented as an ordered skip list. The records within the list are first ordered based on their score and then based on lexicographic order (for records with the same score). The skip list of each bucket provide layers so that you can "skip" over large part of the bucket by simply traversing skip nodes at higher layers. Explicitly, each skip node represents a range, [S_i–Max], where S_i is the score of underlying node at the skip point, and Max is the score of the terminating node in that bucket. The *IX-cache* caches the skip nodes to help shortcircuit the traversal for any key in the range S_i to Max. Note that a hit in the *IX-cache* does not completely eliminate the traversal (as there could be multiple strings with same score) and we have to validate by traversing that portion of the list. Finally, when there are multiple skip nodes from which we can reach a target node, we have to choose which one to cache. Here, we cache the skip node located closest to the median point of the keys as it maximizes reach and short-circuits multiple traversals.

There are two possible deployments of sorted sets based on how the score of each record is set, explicit or implicit: i) An explicit score is assigned by the user if they wish to maintain an ordered collection of items. Examples include retrieving posts in a social network feed based on their popularity, or managing tasks in a priority queue. ii) Alternatively, an implicit score can be calculated as a hash of the string in the record. This enables *METAL* to gracefully support arbitrarily wide string-based keys despite the hardware restricting key bit-widths (e.g., 4–8 bytes). In this case, the scores do not reflect actual ordering, but for organizing records in the skip list and supporting fast lookups. Any hash function can be used to support lookups, but with order-preserving [18] and consistent [21, 44] hashing we can support range scans of strings as well.



Feature	Config
DSA Grid	16 (8×2) —128 (8×16) tiles @1Ghz. Statically configured mesh. 32b/cycle pipelined.
Scratchpad	8K/tile (SpMM) to 64K/tile (JOIN)
Compute	32bit. +,× (256), <<,|,& (256).
Walkers	32 walkers; 4 outstanding (128 total)
Cache	Set-Assoc (16-way): 1 cycle. 7000fJ. X-cache. 1024 entries. 64k. 2 cycles.
IX-cache	Set-Assoc (16-way). 1024 entries. 64k. 5 cycles. 9000fJ/64k.
Memory	HBM 1000, 8 Ch, Bank width: 1024bit. BW:128GB/s
Energy	Reg:50fJ +:210fJ ×:1260fJ Bit:180fJ <<:410fJ
Host	ARMv9, 2Ghz, OOO. 8 way. L1D 64kB; L2 2MB

Figure 14. Overview of Simulation Setup

5 Evaluation

Our toolflow is depicted in Figure 14 using Gem5-Salam [47]. The computation is mapped onto the grid of functional units by our compiler using LLVM [47, 58]. We write a functional model of the target DSA in C/C++ and host code to drive it. The control and dataflow graphs are extracted by lowering the DSA definition to LLVM IR. This is mapped to a runtime model with constrained tile resources. The runtime is a faithful "execute-in-execute" simulator with timing. We implement *IX-cache* as a memory object in Gem5. We use non-coherent crossbars in Gem5 to connect the DSA's components to the scratchpad and *IX-cache*. The DMA engines directly interface with the memory controller. We implemented X-cache [50] as a gem5 cache object. We implement a hardwired controller for miss handlers, and synthesize it from the same walk logic as *IX-cache*. We assume ideal cost for X-cache's handlers i.e., no resource limit and only limited by DRAM access latency. Following a miss, X-cache inserts the leaf into the cache. On a hit, we assume that data is returned on fast path (no additional handler [50]). Fig. 14 lists the simulation parameters. All cache blocks are set to 64 bytes to ensure a fair comparison. We study eight applications across four domain-specific architectures and support five different indexes: B+Trees, Compressed Fibres, Relational tables, R-Trees, and Graphs. Table 2 lists patterns for each DSA and application and the number of ops .

We evaluate two variants, *METAL*-IX and *METAL*. i) ***METAL*-IX** does not use patterns at all; our goal is to showcase *IX-cache* stand-alone efficacy without patterns. *METAL*-IX employs a hardwired eviction policy based on utility. We track utility by using 4-bit saturating counters (one per entry) to track accesses. Other implementations are possible, but index

Table 2. Workload Setup

	Scan[56]	**Sets**[31]	**SpMM**[49]	**Analytics**[56]	**RTree**[57]	**PageRank**[57]
Workload	Random Search		Inner Product	SEL,WHE.,JOIN	Quad. Embedding	PageRank-push
DSA	Gorgon	Gorgon	Capstan	Gorgon	Aurochs	Aurochs
Index	B+Tree	Hash-table	Dyn.Tens. [10, 39]	B+Tree	RTree	Adj. List
Size	10M keys	8M keys	HB/bcs.[1612]	10M records	2 BTrees [10M,300K]	10M nodes
Degree	5 (9 keys)	10	Dynamic (row/col)	5 (9 keys)	BTree-x:5, BTree-y:3	Dynamic (deg.)
Depth	10 levels	-	-	10 levels	BTree-x:10,BTree-y:6	-
Ops/Walk	56 Ops	128 Ops	116 Ops/row	74 Ops	130 Ops	142 Ops
Ops/Compute	6 Ops	48 Ops	111 Ops/row	232 Ops	206 Ops	141 Ops
Index Type	[Lo, Hi]	Id, Key	Row, Col (Int)	[Lo,Hi] (Int)	[Lo,Hi] (Int)	[key, degree](Int)
Pattern	**Level**	**Node**	**Node**	**Level**	**Level+Branch**	**Node+Branch**

probes are infrequent (every few hundred cycles) so counters are sufficient. ii) ***METAL*** employs cache descriptors and dynamically decides which nodes to insert or bypass the *IX-cache*. The pattern remains constant throughout the run, but parameters are updated after a batch of 1 million walks. Table 2 lists the descriptor for each of the workloads.

5.1 Trends: *METAL* vs. Address vs. X-Cache

Here we present an initial investigation on why *METAL*'s cache organization is fundamentally more effective (independent of geometry and policy). We compare *METAL* with a fully-associative address cache with OPT policy (FA-OPT) and state-of-the-art X-Cache [50] which caches leaves. Each cache is configured as 64kB, 1024 entries. We also compare against a 16× larger 1MB address cache. We use three metrics: i) **Miss rate:** The ratio of misses to the number of accesses (Fig. 15). ii) **Working set size:** The fraction of the index touched in the DRAM (Fig. 16), and iii) **Walk latency:** The average walk latency in cycles. (Fig. 17)

❶ **Observation 1: Address-caches are limited by working set; policy has less impact.** FA-OPT has lower miss rate than LRU, but working set still remains high due to repeated root-to-leaf traversals touching $\simeq$ 85% of index. FA-OPT is also 1.8× slower than *METAL*.

❷ **Observation 2: Miss rates can be misleading when comparing cache organizations, since hit and miss path vary.** X-cache may have high miss rate due to leaves, but may still reduce the working set, since hits completely short-circuit and eliminate multiple DRAM accesses. FA-OPT has lower miss rate but has high working set due to repetitive traversals.

❸ **Observation 3: X-cache has high miss rate since leaf working set is high in index data-structures.** X-cache can only cache leaves thus the likelihood of thrashing grows with the number of entries in the index. Furthermore, on misses, X-cache requires complete root-to-leaf walks, which further exacerbates the problem. X-cache miss rate is between 0.6—0.95.

❹ **Observation 4: *METAL* short-circuits more walks, thus reducing the working set size vs. X-cache.** *METAL* caches frequently used intermediate nodes, bypassing low reuse leaf nodes, and ignoring redundant upper nodes. X-Cache only caches the leaf, even though it short circuits effectively, it does not reduce the working set size much because it is oblivious to index access behaviour. The working set size for *METAL* is $\simeq$0.2 vs. $\simeq$0.72 for X-cache.

❺ **Observation 5: METAL reduces walk latency by 1.5× vs. X-cache, 1.8× vs. FA-OPT.** Unlike X-cache, *METAL* does not always cache leaves, rather the workload reuse pattern dictates cache insertion. *METAL* can cache intermediate nodes to improve reach or cache lower nodes to reduce walk latency. FA-OPT does not reduce walk latency, since working set does not reduce and there is no short-circuiting.

❻ **Observation 6: *METAL* shrinks the cache size by 16×.** We include a larger address cache (1MB) to try and match 64k *METAL*'s performance. Even large address caches suffer from inflated working sets that thrash the cache and reduce its effectiveness. A 1MB FA (16× the size of *METAL*) has 20% higher walk latency.

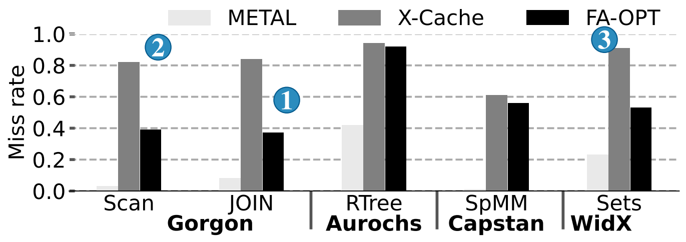


Figure 15. Miss Ratio. 1 = All misses. All caches are 64K.

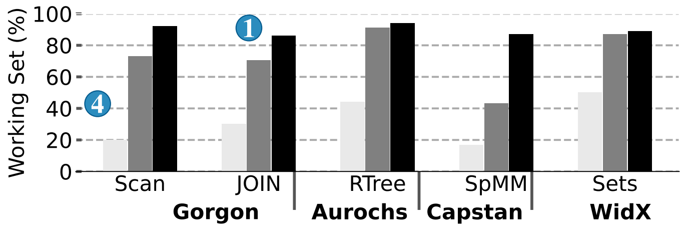


Figure 16. Working Set (lower is better). 100% = All nodes.

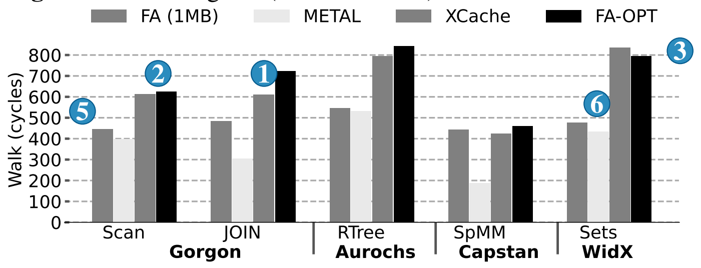


Figure 17. Walk Latency. (lower is better). METAL, X-Cache, FA-OPT are 64K. FA (1MB) is 1MB.

Table 3. Evaluation Summary

Question	Answer
How much can *METAL* improve performance ?	7.8× vs streaming, 4.1× vs. Addr., 2.4× vs. X-cache. § 5.2.
How much DRAM energy can *METAL* save?	1.9× vs streaming. 1.7× vs. Addr., 1.6× vs. X-cache. §5.3.
How much perf. attributed due to *IX-cache*?	5.3× vs. streaming, 2.8× vs. Addr., 1.6× vs. X-cache. §5.2.
How much improvement due to patterns?	1.6×—3.7× over *METAL*-IX. §5.4.
Scalability with large index data-structures?	18 deep, needs 256k *IX-cache*. §5.5 Optimal size: 64k §5.6.
Cache energy	29.5% of total on-chip. upto 5× lower vs. Addr; 3× vs X-cache. § 5.7
Supplemental results (available on hotcrp)	Best geometry: 16-way. 16 banked. Shared vs. Private: Shared is best since access every 70-180 cycles. All experiments use optimal config.

5.2 Performance Evaluation

Result: *Compared to address-cache, METAL short-circuits and reduces working set. Compared to X-Cache, METAL uses patterns to collectively optimize reach and short-circuiting.*

Fig. 18 plots the speedup. In this section and future sections the baseline cache sizes are set to 64k,16-way,16 banks (see §5.6 for design sweep). *METAL*-IX and *METAL* achieve speedup by short-circuiting and maximizing reach. *METAL* performs better than *METAL*-IX since we enable patterns to capture the reuse of the algorithm, and cache nodes with maximal benefit. ① In workloads with significant working set size - JOIN, *METAL* shows ≊4× improvement compared to address cache. There are 2 reasons for this: i) *IX-cache* alone improves by 2.6 × due to short-circuiting and saving multiple DRAM accesses, whereas a hit in the address cache only eliminates a single access. ii) *METAL* introduces patterns to further improve 4 × (analysed in detail:§ 5.4). We only improve by 4×, since it has high arithmetic intensity : 318 ops/walk. We improve performance over X-Cache [50] by 2.4 ×, since we enable caching of intermediate nodes, adapting to the search batch of keys. ② We maximise performance in workloads where reach is important (Scan, JOIN) compared to X-Cache, since it only caches leaves. In RTree, working sets overflow and gains are limited.

Shallow vs. Deep Indexes: SpMM and Sorted Sets (Sets) can be configured to use shallow index data-structures (-S in plot) or deep indexes. SpMM-S employs shallow fibers [35], while SpMM employs deep sparse tensors [10]. Sets-S mimics a low associativity hash-table with $\simeq 10^3\times$ than Sets (deep version) uses longer lists in each bucket. ③ *METAL* improves by 2.4× vs. X-cache for deep indexes. We cache high reuse leaves and short-circuit the entire walk. X-cache also short-circuits but prematurely evicts. *METAL* sets lifetime using descriptor to prevent premature evictions. In the shallow versions (-S in plot) *METAL*'s behavior is similar to X-cache; only 15% improvement. This is because there is less opportunity for exploiting reach. *METAL*-IX does not include patterns and pays the penalty for higher cache hit latency and cache pollution.

5.3 DRAM Energy

Fig. 19 shows the normalized dynamic energy in DRAM. Even in workloads where speedup is limited, *METAL* reduces DRAM Energy (e.g., $\simeq 2\times$ in Nest.SEL and WHERE). *METAL* reduces the the total number of access by short-circuiting walks. In memory-bound workloads (Scan, SpMM in Fig. 19:④), we see maximum reduction of DRAM energy. X-Cache's hit path maximizes short-circuiting to the leaves. Thus, X-Cache has minimal traffic benefit over the address cache. *METAL* caches intermediate nodes, maximizes reach and saves traffic. SpMM exhibits high short-term reuse. *METAL*'s pattern is effective at capturing the leaf reuse and ensuring that walks are completely short-circuited. In JOIN, *METAL* experiences high contention as it targets multiple B+Trees. *METAL* short-circuits less and hence achieves less traffic reduction. For the shallow versions (-S) we save less since leaf node reuse can be captured by all designs; we still save 10–15% since patterns prevent premature evictions.

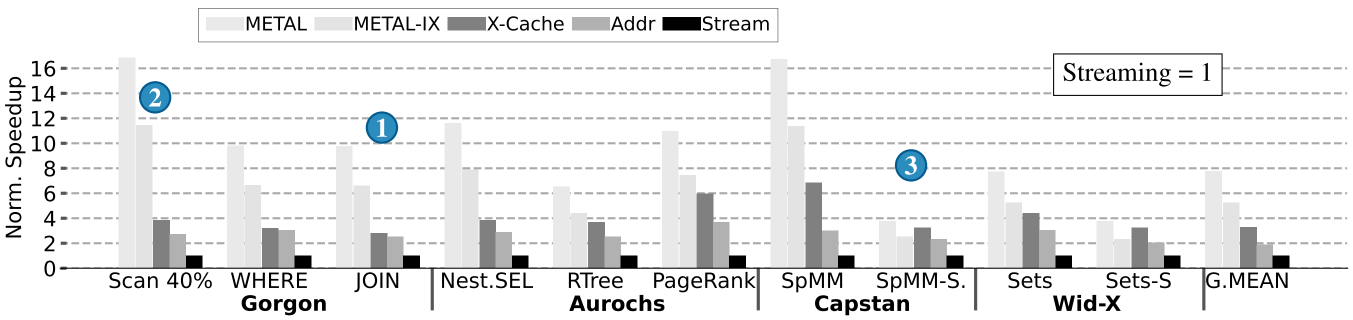


Figure 18. Speedup. *METAL* vs. X-Cache vs. Address vs. Stream. (higher is better). -S: Shallow Fibers.

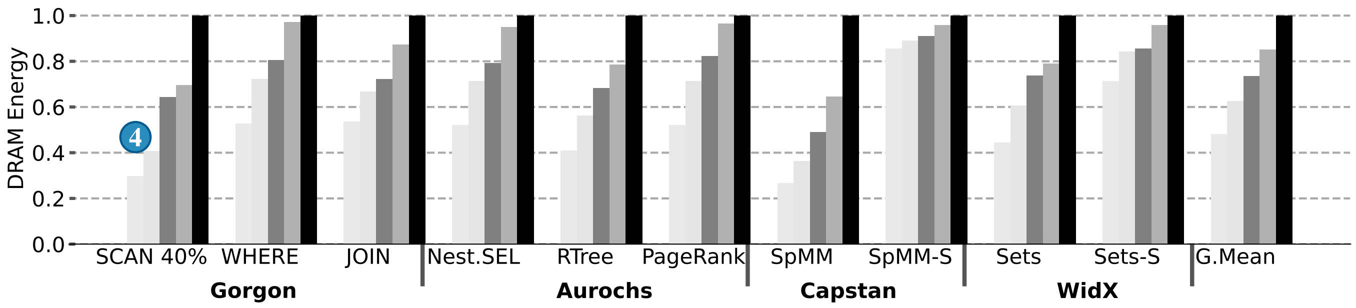


Figure 19. Normalized DRAM Energy (lower is better).

5.4 METAL Speedup Breakdown

Result: *Patterns avoid misbehavior of hardwired policy of IX-cache and improves performance further by 1.5—4×; dynamically tuning parameters further improves performance by 10—30%.*

METAL's performance improvement is due to three factors: *IX-cache* that short-circuits walks, patterns that capture the reuse of the algorithm, and parameters that dynamically adjust regions that are cached. We now breakdown the contribution of each factor (Fig. 20). Overall, the *IX-cache* hardware alone improves performance between 3–8× vs. streaming. Further improvement is limited by the hardwired policies that are reactive (e.g., LFU or LRU) to manage locality. Introducing patterns improves speedup further to 3.5–14x; 1.5–2× improvement vs. *IX-cache*: i) **Patterns prevent thrashing:** Hardwired *IX-cache* greedily caches all nodes touched during a walk. Patterns explicitly set margins below which nodes that are not frequently used will be bypassed and not cached. ii) **Patterns avoid redundant insertions:** In workloads with high data reuse, upper levels in the index are strictly redundant. *IX-cache* does not realize this and wastes space on redundant index nodes. Patterns explicitly target lower leaves only thereby improving cache utilization e.g., in SpMM we improve performance by 4 ×. iii) **Patterns exploit cooperative caching and maximize reach.** Patterns specify a band of index levels that contain common roots across walks. In applications with deep indexes (e.g., Trees in SpMM, Lists in Pagerank) these cached common roots short-circuit many walks. Dynamically tuning parameters further improve performance 10—30% over patterns (overall 3.8—16× speedup vs. streaming). Parameters dynamically redraw the index region being cached by varying the start-to-end band of levels, and left-to-right sub-branches, based on the keys touched. Overall impact is high on workloads where keys are fetched vary frequently (e.g., Scans, Join, R-tree).

How do patterns adaptively cache? To further dissect the reasons for speedup, we investigate the contents of the *IX-cache* (Fig. 21) (with and without patterns). We even fine-tuned the standalone *IX-cache* based on a profile run. Through this, we bypass the levels to prevent thrashing. Despite this, patterns and parameter tuning achieve better cache behavior. There are two ways in which patterns and parameters improve: i) **Adapting to target reach (e.g., Scan):** *IX-cache* alone spreads the cache capacity across multiple levels of the index. Tuning the pattern parameters narrows the levels cached and dynamically finds the best [start—end] level for maximizing reach. ii) **Adapting for leaf reuse (e.g., SpMM):** *IX-cache* tends to waste space on intermediate and upper nodes, that are redundant when leaf nodes are cached. The reuse pattern helps bypass the intermediate nodes and straightaway targets the leaf. SpMM-S has index < 4 levels, therefore, the cache occupancy shows only 1-3 cached. We provide a plot illustrating how the parameter tuning help *METAL* adapt over time

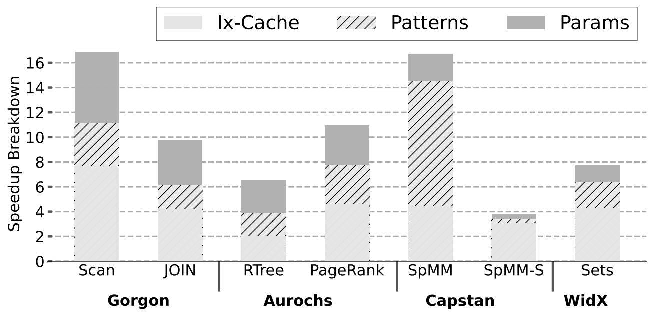


Figure 20. Breakdown of factors contributing to *METAL*'s speedup. IX: *IX-cache* only (with hardwired policy). Patterns: Reuse managed with static parameters. Params: Reuse managed with dynamic parameter tuning.

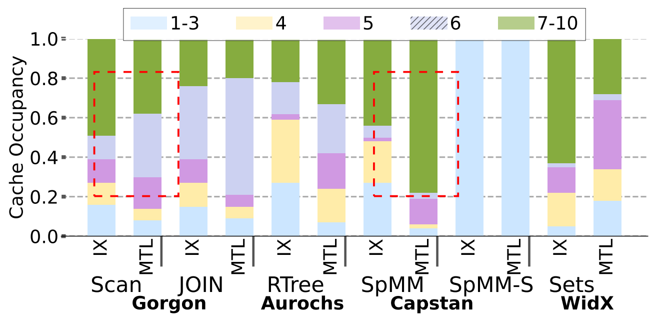


Figure 21. *METAL*-IX vs. *METAL* (MTL in X-axis). In Sorted Sets, the skip list can be arbitrarily deep; so we split into 10 regions (1: head of skip list). SpMM-S: Fibers are 3 levels.

(Fig. 22). We show a total of 10 million walks of execution split into windows of 1 million. For the Scan we can see that parameters are able to find the levels best suited for walks within each window while the fixed pattern is unable to adapt.

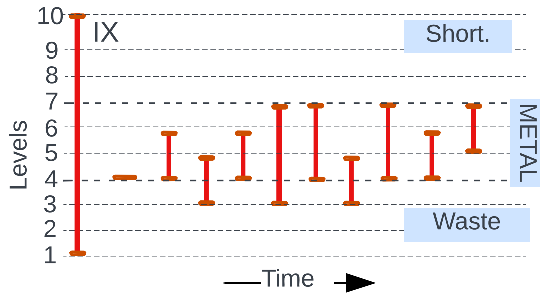


Figure 22. Level pattern adaptivity with parameter tuning. Y-axis: Levels cached as walks change. X-axis. 1 million walks. *IX-cache* chooses hard wired policy. Benchmark: Scan

5.5 *METAL* vs. Index Size

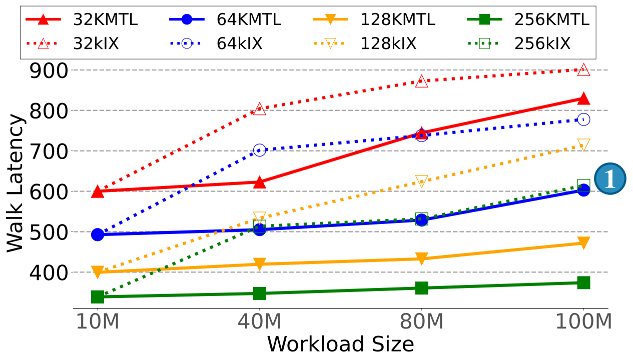
Result: *METAL can use patterns to scale up and support larger datasets, without requiring a larger IX-cache. Increasing records from 10M to 100M introduces only 15% penalty.*
Result: *A 40× increase in database size requires only 8× increase in IX-cache size.*

Two separate factors determine index size: i) Number of records, ii) index depth. First, we vary the B+Tree from 10M to 100M records and *IX-cache* size from 32kB - 256kB (512 - 4096 entries). We show *METAL*-IX and *METAL* scaling with increasing index size. Fig. 23a shows the average walk latency. ❶ We find that *METAL* adapts to larger databases without increasing cache size due to patterns which allow for optimizing space in the *IX-cache*. Thus, for JOIN, *METAL* caches common intermediate nodes, focuses the cache on fewer levels, and maximizes the number of walks benefiting from short-circuiting.

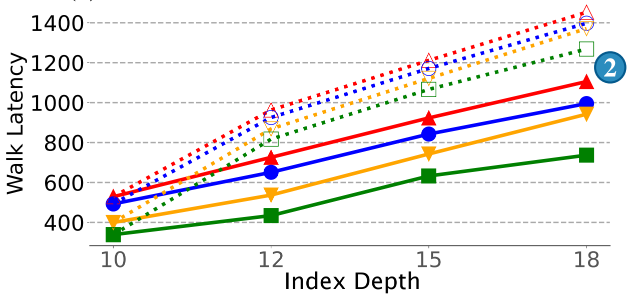
Secondly, we varied the index depth from 10-18 levels for *METAL*-IX and *METAL* (Fig. 23b). This increases the database size by 40x (10M to 400M records). We find that the walk latency increases by 2 × for *METAL* and 3 × for *METAL*-IX , due to traversing more nodes per walk. We also observe that *METAL*-IX 's performance degrades at a faster rate due to inefficient reuse region capture. ❷ Through *METAL* , we reduce the *IX-cache* size by 8 × because 32KB *IX-cache* with *METAL* scales better than 256kB *IX-cache* with *METAL*-IX .

5.6 Design Sweep

Result: *A 64KB IX-cache can support up to 64 compute tiles while consuming less than 50% HBM bandwidth.*
Result: *In SpMM a 16K cache is sufficient, since workload has immediate reuse. In workloads with varied index reuse METAL supports larger DSA e.g., JOIN. In RTree workloads with large working set, DSA is bandwidth limited*



(a) How does *METAL* scale with index size.



(b) How does METAL scale with index depth

Figure 23. Cache size vs. Index size. Legend: **MTL:** *METAL*. **IX:** *METAL*-IX. workload used: JOIN

We study the scaling in performance on varying the tile count from 16-128 and *IX-cache* size from 8kB-2MB. Our goal is to find the Pareto-optimal design point for *METAL* . We only show *IX-cache* size up to 256kB, since the DSA is parallelism (par.) limited. Fig 24 plots for workloads with different reuse patterns: JOIN, SpMM and RTree. We vary the *IX-cache* size on the x-axis and plot the normalised Speedup on the y-axis. The plot is normalised to 8-tile streaming DSA. Within each plot, we classify the performance curves into 3 regions: i) Bandwidth Limited (Band. Lim.): In this region, the DSA consumes $\geq$ 50% of HBM peak bandwidth, and has high miss rate. ii) Cache Limited (Cache Lim.): In this region, the *IX-cache* size and policy (or pattern) can influence miss rate. iii) Parallelism Limited (Par. Lim.): In this region, the performance is dependent on the number of compute tiles.

5.7 Energy

Result: *METAL reduces cache energy by short-circuiting walks, and reducing number of accesses. 3× lower energy.*

Fig, 25:Top compares the energy of cache organizations. Energy = per-access cost × #accesses. The baseline is a 16-way address cache with the data array accessed only on a match. *METAL* 's tags are also stored in SRAM, the only difference is the range match. We find that the total per-access energy is more expensive for *METAL* - 9000fJ vs 7000fJ (for X-cache and address-cache). Compared to the address cache, *METAL* reduces total accesses by 2-4 ×. Compared to X-Cache, *METAL* achieves higher hit rate by targeting high

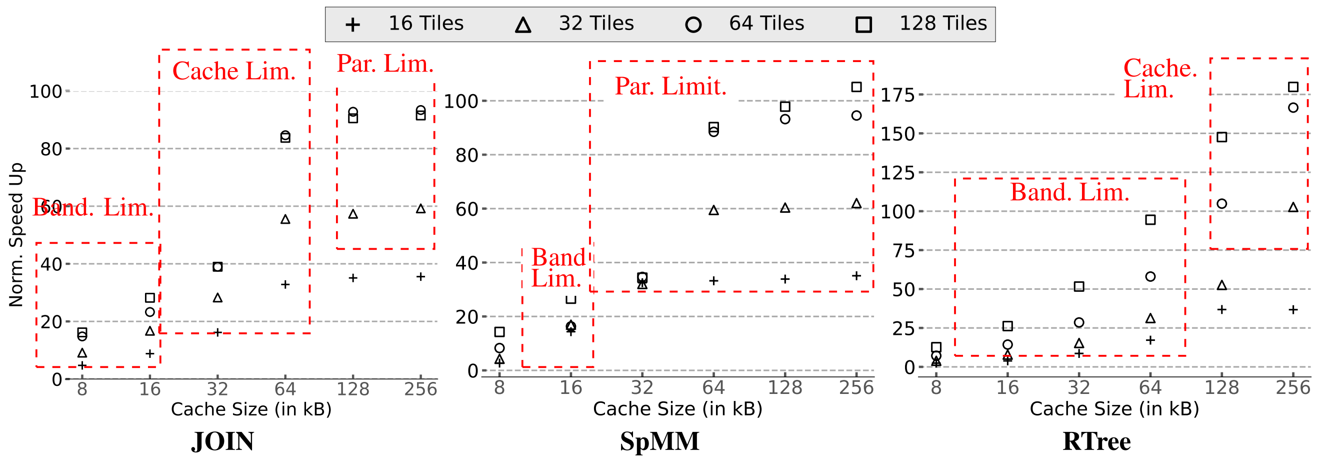


Figure 24. Y-axis: Normalized Speedup. X-axis: Cache size. Legend: 16-128 compute tiles. Base: 8-tile streaming.

reuse index nodes, not just leaves. We observe that the *IX-cache* is queried on an average every 108 cycles. This makes the accesses to the *IX-cache* sparse and reduces total access cost compared to address cache models where every memory access needs to go through the cache hierarchy.

Fig. 25:Bottom, breaks down the energy of different modules: compute tile, *IX-cache*, walker logic + pattern controller. We show representative workloads from each of the DSAs. The *IX-cache* accounts for $\frac{1}{3}$ of overall on-chip energy.

6 Supplemental Related Work

Section 2 gave a detailed comparison against start-of-the-art. The long history of work on decoupled DMA for shuttling dense tensors includes scatter/gather engines [20, 23, 28, 51], memory controllers [8], and tiled DMAs [1, 12, 40]. They only target streaming. Stream-DSAs have targeted indexed data-structures e.g., Aurochs [57] and Stream-join [14]. Stream-DSAs do not recognize reuse in index data-structures, so they suffer significant bandwidth penalty in workloads with reuse. Aurochs [57] expresses walks as a sequence of functional patterns. Stream-join [14] expresses walks using indirectly-indexed patterns.

There has also been work on walkers in FPGAs [9, 12], CPUs [31], and CGRAs [57]; Widx [31] stored data in an address-based cache, Ax-DAE [9] stored the data in a scratchpad, DAS-X [33] used an object cache that captures leaf reuse, similar to X-cache. We demonstrate that leaf data reuse may be lower in index data-structures. There has also been extensive work on hardware prefetching for linked lists [17, 29, 48] which focused on the walk and traversal, not on index reuse.

There has been work that targeted explicit reuse in tiled algorithms [1, 12]. Buffets [40] created a portable library for managing scratchpads. There have been proposals for remapping in address-caches (Stash [32], Jenga [53], and Hotpads [54]). These target affine-loops and require the DSA to define the loop order, and access stride.

Finally, *IX-cache* generalizes the classical concept of guarded page tables and translation caches [5, 6, 34, 59]. This paper targets DSAs, while CPU/GPU extensions are future work.

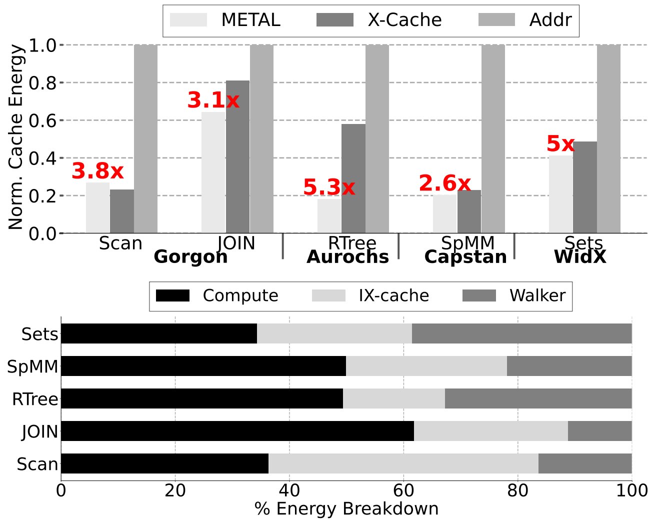


Figure 25. Top: Cache Energy. (Red: Cache access reduction relative to address-cache.) **Bottom:** Energy Breakdown.

7 Conclusion

We have developed *METAL*, an architectural template to enable DSAs to manage and reuse index data-structures. There are two complementary ideas in *METAL*: 1) *IX-cache*, a novel cache architecture that uses key indexes as cache tags for short circuiting index walks. ii) Reuse Patterns, an abstraction to manage *IX-cache* locality at the granularity of the indexes. *METAL* will help "generalize" DSAs and enable them to target a wider range of applications.

Acknowledgments

We would like to express our sincere appreciation to the reviewers and our shepherd, Peter Desnoyers, for their invaluable feedback and guidance throughout the review process. Their constructive input and insights played a pivotal role in enhancing the quality and clarity of the paper.

References

[1] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. 2011. Leap Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic. In Proc. of the 19th FPGA (FPGA '11). New York, NY, USA.

[2] Md Nur Ahmed. 2024. https://dev.to/mdnurahmed/simple-scalable-search-autocomplete-systems-1j18.

[3] Tutu Ajayi, Vidya A Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B Kahng, Minsoo Kim, Jeongsup Lee, Uday Mallappa, Marina Neseem, et al. 2019. Toward an open-source digital flow: First learnings from the openroad project. In Proc. of the 56th Annual Design Automation Conference 2019. 1–4.

[4] Daehyeon Baek, Soojin Hwang, Taekyung Heo, Daehoon Kim, and Jaehyuk Huh. 2021. InnerSP: A Memory Efficient Sparse Matrix Multiplication Accelerator with Locality-Aware Inner Product Processing. In 30th Int'l. Conf. on Parallel Architectures and Compilation.

[5] Thomas W Barr, Alan L Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don't Walk (the Page Table). In Proc. of the 37th ISCA.

[6] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating Two-Dimensional Page Walks for Virtualized Systems. SIGOPS Oper. Syst. Rev. 42, 2 (mar 2008), 26–35.

[7] N. V. Vijaya Krishna Boppana and Saiyu Ren. 2016. A Low-Power and Area-Efficient 64-Bit Digital Comparator. J. Circuits Syst. Comput. 25, 12 (2016).

[8] J. Carter, W. Hsieh, L. Stoller, M. Swanson, Lixin Zhang, E. Brunvand, A. Davis, Chen-Chi Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. 1999. Impulse: building a smarter memory controller. In Proceedings Fifth International Symposium on High-Performance Computer Architecture.

[9] Tao Chen and G Edward Suh. 2016. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling.. In Proc. of the 49th MICRO. 1–12.

[10] Stephen Chou and Saman Amarasinghe. 2022. Compilation of dynamic sparse tensor algebra. Proc. of the ACM on Programming Languages 6, OOPSLA2, 1408–1437.

[11] Chang Chua and R.B.N. Kumar. 2017. An Improved Design and Simulation of Low-Power and Area Efficient Parallel Binary Comparator. Microelectron. J. (aug 2017), 84–88.

[12] Eric S Chung, James C Hoe, and Ken Mai. 2011. CoRAM: an in-fabric memory architecture for FPGA-based computing. In Proc. of the 19th FPGA.

[13] Jason Clemons, Chih-Chi Cheng, Iuri Frosio, Daniel R Johnson, and Stephen W Keckler. 2016. A Patch Memory System for Image Processing and Computer Vision.. In Proc. of the 49th MICRO. 1–13.

[14] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards general purpose acceleration by exploiting common data-dependence forms. In Proc. of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. 924–939.

[15] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms.. In Proc. of the 52nd MICRO. 924–939.

[16] William J. Dally, Yatish Turakhia, and Song Han. 2020. Domain-Specific Hardware Accelerators. Commun. ACM 63, 7 (June 2020), 48–57.

[17] E Ebrahimi, O Mutlu, and Y N Patt. 2009. Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems. In Proc. of the 15th HPCA.

[18] Edward A Fox, Qi Fan Chen, Amjad M Daoud, and Lenwood S Heath. 1991. Order-preserving minimal perfect hash functions and information retrieval. ACM Transactions on Information Systems (TOIS) 9, 3 (1991), 281–308.

[19] Fabio Frustaci, Stefania Perri, Marco Lanuzza, and Pasquale Corsonello. 2012. Energy-efficient single-clock-cycle binary comparator. Int. J. Circuit Theory Appl. 40, 3 (2012), 237–246.

[20] Daichi Fujiki, Niladrish Chatterjee, Donghyuk Lee, and Mike O'Connor. 2019. In Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis. New York, NY, USA, Article 55.

[21] A González-Beltrán, Peter Milligan, and Paul Sage. 2008. Range queries over skip tree graphs. Computer Communications 31, 2 (2008), 358–374.

[22] Goetz Graefe et al. 2011. Modern B-tree techniques. Foundations and Trends® in Databases 3, 4 (2011), 203–402.

[23] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. 2007. Efficient gather and scatter operations on graphics processors. In SC '07: Proc. of the 2007 ACM/IEEE Conference on Supercomputing. 1–12.

[24] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. Commun. ACM 62, 2 (Jan. 2019), 48–60.

[25] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. 2015. Efficient execution of memory access phases using dataflow specialization.. In Proc. of the 42nd ISCA. 118–130.

[26] Ibrahim Kamel and Christos Faloutsos. 1992. Parallel R-trees. ACM SIGMOD Record 21, 2 (1992), 195–204.

[27] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. 2019. SMASH: Co-designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations. In Proc. of the 52nd MICRO.

[28] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. 2003. Programmable Stream Processors. Computer 36, 8 (aug 2003), 54–62.

[29] M Karlsson, F Dahlgren, and P Stenstrom. 2000. A Prefetching Technique for Irregular Accesses to Linked Data Structures. In Proc. of the 6th HPCA.

[30] Michael S Kester, Manos Athanassoulis, and Stratos Idreos. 2017. Access path selection in main-memory optimized data systems: Should I scan or should I probe?. In Proc. of the 2017 ACM International Conference on Management of Data. 715–730.

[31] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin T Lim, and Parthasarathy Ranganathan. 2013. Meet the walkers: accelerating index traversals for in-memory databases.. In Proc. of the 46th MICRO. 468–479.

[32] Rakesh Komuravelli, Matthew D Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V Adve, and Vikram S Adve. 2015. Stash: have your scratchpad and cache it too.. In Proc. of the 42nd ISCA. 707–719.

[33] Snehasish Kumar, Naveen Vedula, Arrvindh Shriraman, and Vijayalakshmi Srinivasan. 2015. DASX: Hardware accelerator for software data structures. Proc. of the International Conference on Supercomputing 2015-June (2015), 361–371.

[34] Jochen Liedtke and Kevin Elphinstone. 1996. Guarded Page Tables on Mips R4600 or an Exercise in Architecture-Dependent Micro Optimization. SIGOPS Oper. Syst. Rev. 30, 1 (jan 1996).

[35] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In Proc. of the 29th ACM on International Conference on Supercomputing. 339–350.

[36] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Prasant Singh Rawat, Sriram Krishnamoorthy, and P. Sadayappan. 2019. An Efficient Mixed-Mode Representation of Sparse Tensors. In Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis.

[37] Tayo Oguntebi and Kunle Olukotun. 2016. Graphops: A dataflow library for graph analytics acceleration. In Proc. of the FPGA.

[38] Oracle. [n. d.]. Scans. https://databaseinternalmechanism.com/oracle-database-internals/index-lookup-unique-scanrange-scan-full-scan-fast-full-scan-skip-scan/.

[39] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. 2021. Terrace: A hierarchical graph container for skewed dynamic graphs. In Proc. of the 2021 International Conference on Management of Data. 1372–1385.

[40] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Clayton Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W Keckler, Christopher W Fletcher, and Joel S Emer. 2019. Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration.. In Proc. of the 24th ASPLOS. 137–151.

[41] Stefania Perri and Pasquale Corsonello. 2008. Fast Low-Cost Implementation of Single-Clock-Cycle Binary Comparator. IEEE Transactions on Circuits and Systems II: Express Briefs 55, 12 (2008), 1239–1243.

[42] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matthew Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Paterns.. In Proc. of the 44th ISCA. 389–402.

[43] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In IEEE International Symposium on High Performance Computer Architecture (HPCA).

[44] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M Hellerstein, and Scott Shenker. 2004. Prefix hash tree: An indexing data structure over distributed hash tables. In Proc. of the 23rd ACM symposium on principles of distributed computing, Vol. 37. St. John's Newfoundland, Canada.

[45] Redis. 2024. https://github.com/redis/redis/blob/unstable/src/t_zset.c.

[46] Redis. 2024. https://redis.com/glossary/redis-sorted-sets/.

[47] Samuel Rogers, Joshua Slycord, Mohammadreza Baharani, and Hamed Tabkhi. 2020. Gem5-SALAM: A System Architecture for LLVM-Based Accelerator Modeling.. In Proc. of the 53rd MICRO. 471–482.

[48] Amir Roth, Andreas Moshovos, and Gurindar S Sohi. 1998. Dependence Based Prefetching for Linked Data Structures. In Proc. of the 8th ASPLOS.

[49] Alexander Rucker, Matthew Vilim, Tian Zhao 0001, Yaqi Zhang 0001, Raghu Prabhakar, and Kunle Olukotun. 2021. Capstan: A Vector RDA for Sparsity.. In Proc. of the 54th MICRO. 1022–1035.

[50] Ali Sedaghati, Milad Hakimi, Reza Hojabr, and Arrvindh Shriraman. 2022. X-cache: a modular architecture for domain-specific caches. In Proc. of the 49th Annual International Symposium on Computer Architecture. 396–409.

[51] Vivek Seshadri, Thomas Mullins, Amirali Boroumand, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2015. Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-Unit Strided Accesses. In Proc. of the 48th International Symposium on Microarchitecture (Waikiki, Hawaii) (MICRO-48). Association for Computing Machinery, New York, NY, USA, 267–280.

[52] Smrchy. [n. d.]. https://github.com/smrchy/redis-tagging.

[53] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Jenga: Software-Defined Cache Hierarchies. In 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). 652–665.

[54] Po An Tsai, Yee Ling Gan, and Daniel Sanchez. 2018. Rethinking the memory hierarchy for modern languages. Proc. of the Annual International Symposium on Microarchitecture, MICRO 2018-Octob (2018), 203–216.

[55] Piyush Tyagi and Rishikesh Pandey. 2020. High-Speed and Area-Efficient Scalable N-bit Digital Comparator. IET Circuits, Devices & Systems 14, 4 (2020), 450–458.

[56] Matthew Vilim, Alexander Rucker, Yaqi Zhang 0001, Sophia Liu, and Kunle Olukotun. 2020. Gorgon: Accelerating Machine Learning from Relational Data.. In Proc. of the 47th ISCA.

[57] Matthew Vilim, Alexander Rucker, and Kunle Olukotun. 2021. Aurochs: An Architecture for Dataflow Threads.. In Proc. of the 48th ISCA.

[58] Jian Weng, Jian, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. 2020. DSAGEN: Synthesizing Programmable Spatial Accelerators.. In Proc. of the 47th ISCA. 268–281.

[59] Idan Yaniv and Dan Tsafrir. 2016. Hash, Don't Cache (the Page Table). SIGMETRICS Perform. Eval. Rev. 44, 1 (jun 2016), 337–350.

[60] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sánchez. 2021. Gamma: leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In Proc. of the 26th ASPLOS.

[61] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. SpArch: Efficient architecture for sparse matrix multiplication. In In Proc. of 26th HPCA.

[62] ZhangYunHao. 2024. https://github.com/zhangyunhao116/skipset.