

RANGE-BLOCKS: A Synchronizing Facility for Data-Structures in Domain-Specific Architectures

by

Anagha Molakalmur Anil Kumar

B.Tech., PES University, 2021

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© **Anagha Molakalmur Anil Kumar 2024**
SIMON FRASER UNIVERSITY
Summer 2024

Copyright in this work is held by the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.

Declaration of Committee

Name: Anagha Molakalmur Anil Kumar

Degree: Master of Science

Thesis title: RANGE-BLOCKS: A Synchronizing Facility for Data-Structures in Domain-Specific Architectures

Committee:

Chair: Anders Miltner
Assistant Professor, Computing Science

Arrvindh Shriraman
Supervisor
Associate Professor, Computing Science

Alaa Alameldeen
Committee Member
Associate Professor, Computing Science

Tianzheng Wang
Examiner
Assistant Professor, Computing Science

Abstract

The slowdown of Moore’s law and Dennard scaling has emphasized the importance of domain-specific architectures. These architectures improve compute performance by leveraging architectural specialization to achieve efficiency and performance gains to handle specific tasks and applications, that traditional scaling could no longer provide. Current domain-specific architectures (DSAs) work with static data structures, only supporting in-place updates (updates that don’t require data structure modifications). As DSAs target real-world applications, supporting mutable and dynamically resizable data structures becomes necessary. DSAs lack a synchronization facility, so they cannot support dynamic data structures and are forced to use address-based atomics or batch updates on the host. Both approaches introduce prohibitive performance penalties, requiring large lock caches. Rangeblocks (RBlox) develops a hardware synchronization facility for DSAs to support dynamic data structures, using key ranges to capture synchronization boundaries, tapping into the inherent parallelism of the data structure layout. We make two novel contributions, along with hardware implementation: i) Range locks are symbolic, compactly representing mutexes on multiple nested objects. Thus, any insert requires a single range lock, and a small on-chip table suffices (2kb) compared to large caches (256kb) for address-based locks. ii) Ranges explicitly represent regions of interest, instantly achieving mutual exclusion. On a 128-tile dataflow DSA, we improve performance by 15 \times , reduce DRAM bandwidth by 4 \times , save 70% of on-chip traffic, and require 6.6% of on-chip energy (we demonstrate scalability up to 256 tiles).

Keywords: Synchronization facility; DSA; Hardware Support; Dynamic Data Structures

Dedication

To Aditya, Harshitha, Amma, Bapa, Aiji, Thatha and in loving memory of Ammamma.

Acknowledgements

I am deeply grateful for all the support and encouragement from my supervisor, Dr. Arvindh Shriraman, throughout my master's program. I would also like to thank all my lab mates for their support, with special thanks to Aditya Prasanna, Milad Hakimi, and Ali Sedaghati. Finally, I would like to express my gratitude to my family for their tremendous encouragement and unyielding support.

Table of Contents

Declaration of Committee	ii
Abstract	iii
Dedication	iv
Acknowledgements	v
Table of Contents	vi
List of Tables	viii
List of Figures	ix
1 Introduction	1
1.1 Mutations on host	2
1.2 Range-Blocks overview	3
1.3 Contributions	3
1.4 Publications	5
1.5 Dissertation Outline	6
2 Impact and Scope	7
2.1 Indexes in target DSAs	7
2.2 Architecture Scope: Spatial Dataflow Architectures.	9
2.2.1 Enabling Associative <i>data-structures</i> in DSAs	10
2.3 Limitations of Competing Approaches	11
3 Range-Blocks	13
3.1 Locking Fundamentals: A B+Tree Example	13
3.2 Overview of <i>RBlox</i> Tables	15
3.3 Execution Models	16
3.4 Why range locks are more efficient in hardware?	18
3.5 Table Microarchitecture	19
3.6 Range Lock API	20

3.7	<i>RBlox</i> Invariants and Correctness	21
4	<i>RBlox</i> Evaluation	23
4.1	Methodologies	23
4.2	Evaluation	25
4.2.1	Performance Evaluation	25
4.2.2	Scalability Study	26
4.2.3	Energy	27
4.2.4	UTable size vs. # of tiles in DSA	28
4.2.5	Impact of Update ratio	29
5	METAL	30
5.1	Index Cache	30
5.1.1	Walk Pipeline	32
5.1.2	Similarities and differences between <i>IX-cache</i> and <i>RBlox</i> tables: . . .	33
5.2	<i>METAL-IX</i> Evaluation	34
5.2.1	Performance	35
5.2.2	DRAM Energy	36
5.2.3	Energy	36
5.2.4	Set-Associative vs. Fully-Associative IX-Cache	37
5.2.5	Shared vs. Private <i>IX-cache</i>	39
6	Summary & Thoughts	40
	Bibliography	41

List of Tables

Table 3.1	<i>RBlox</i> vs Lock Caches.	18
Table 4.1	Evaluation Summary	24
Table 5.1	Overview of Simulation Setup	35

List of Figures

Figure 1.1	Address-based Locks vs. Range-blocks synchronization for a B+tree traversal. For clarity, we use an explicit notation for ranges: Lo (leftmost leaf) and Hi (rightmost leaf) leaf. Inf: Max leaf value in the tree.	2
Figure 1.2	Dissertation overview	6
Figure 2.1	index walk with address cache vs. with <i>IX-cache (METAL-IX)</i> . . .	8
Figure 2.2	Traits of Associative Containers.	10
Figure 2.12	Prior State-of-the-Art Approaches to Synchronizing <i>dynamic data-structures</i>	11
Figure 3.1	Illustration of synchronization in a B+tree. [15–31] is assumed to have sufficient space for expansion. For clarity, we use an explicit notation for each node and list the Lo (leftmost leaf) and Hi (rightmost leaf) keys across sub-branches.	13
Figure 3.2	Percentage of safe nodes in a B+tree by level. B+Tree depth: 10. # size: 10M nodes. Degree: 5	14
Figure 3.3	Range block table and fields.	14
Figure 3.6	Illustration of <i>RBlox</i> . Inserts allocate ranges in the LTable; reads self-validate. [15–31] is assumed as safe.	15
Figure 3.7	Illustration of <i>RBlox++</i> . Inserts use unlocked safe ranges in UTable to instantly grab locks to establish a mutation zone.	15
Figure 3.8	Illustration of lock-free reads.	18
Figure 3.9	Organizing <i>RBlox</i> table into banks and sets	19
Figure 3.10	Right: Synthesis of tag match logic in Nangate 45nm. depth = 10, entries = 256. Left: Floorplan	20
Figure 3.11	<i>RBlox</i> Correctness Illustration	22
Figure 4.1	System Configuration	24
Figure 4.2	Normalized Speedup. Higher is better. Configuration — 20% Updates, DSA:128 tiles. LTable:1 entry/tile. UTable: 4 banks×128 sets×8 ways (4096).	25

Figure 4.3	Speedup with increasing DSA tiles from 16 to 128. Normalized to 16T, Base.	26
Figure 4.4	DRAM Energy. Y-axis: Normalized to LCache.	27
Figure 4.5	Energy Pie. (<i>RBlox++</i>) LTable: 128 entries (4 banks×32). UTable: 4096 entries. 4banks×128sets×8 ways.	27
Figure 4.6	Impact of hardware concurrency (#Tiles) vs. UTable size. X-axis: UTable size. Y-axis: Speedup. Baseline: 16T. Red line: <i>RBlox</i> 64 Tiles	28
Figure 4.7	Impact of update ratio. LTable: 32, UTable: 4096 (4 banks, 128 sets). Benchmark: Database.	29
Figure 5.1	<i>METAL-IX</i> micro-architecture.	30
Figure 5.2	Packing index nodes into cache blocks	31
Figure 5.3	Hit Path: <i>IX-cache</i> Logic	32
Figure 5.4	Set-associative IX-Cache	33
Figure 5.5	Cache miss handler.	33
Figure 5.6	<i>METAL</i> simulation set up.	34
Figure 5.7	Speedup. <i>METAL-IX</i> vs. X-Cache vs. Address vs. Stream. (higher is better).	35
Figure 5.8	Normalized DRAM Energy (lower is better).	36
Figure 5.9	Top: Cache Energy. (Red: Cache access reduction relative to address-cache.) Bottom: Energy Breakdown.	37
Figure 5.10	<i>IX-cache</i> Associativity. Left Y: Exe. Time Right Y: Miss rate. Lower is Better. Baseline: Fully Assoc.	37
Figure 5.11	Impact of key block size. Baseline = 256 wide.	38
Figure 5.12	Impact of reducing <i>IX-cache</i> entry size.	38
Figure 5.13	Private. vs Shared <i>IX-cache</i> . Lower is better.	39

Chapter 1

Introduction

Traditional general purpose processors (GPP) face limitations and bottlenecks with the slowdown of Moore’s law and Dennard scaling, which push for higher compute capabilities, maintaining the size and power consumption. Innovations in hardware design have pushed for a more significant performance boost with the development of hardware accelerators or domain-specific architectures (DSAs). These specialized architectures reduce the cost and power consumption for application-specific workloads compared to GPPs.

The data explosion, with the proliferation of data generated, stored, and processed, represents one of the most significant challenges in the modern digital era, with an increasing demand for high-performance and energy-efficient computing.

GPPs struggle to provide efficient processing capabilities required to process these enormous amounts of data. Domain-specific architectures (DSAs) deviate from general-purpose processors and cater to tailored workload-based optimizations, providing a platform for processing data in parallel and offering scalability with higher performance and efficiency.

Current state-of-the-art spatial dataflow DSAs cater to data analytics, machine learning, database workloads, etc [59, 60, 52]. However, they only deal with statically assigned data structures at runtime and do not offer a facility to work with dynamically mutating data structures. Many real-world applications such as graph processing [36, 56, 66, 3], machine learning [49, 42], and data analytics [17] involve dataset changes over time and often frequently. The challenge is that dataset changes necessitate changes to the underlying data structure, e.g., new vertex insertions in a graph or new rows in a database. We refer to data structures that effectively support such changes as *dynamic data structures (DDX)* [11, 41, 6, 32]. Unfortunately, current state-of-the-art DSAs do not support mutations; they only work with static data-structures [39, 60, 14, 30, 2, 46, 13, 22, 15, 25, 64]. Static data-structures [34, 28, 8] are typically affine, and mutations require moving already-stored data in memory. We find that the updates stall readers and prior work have shown that updates account for up to 90% of time [35, 37].

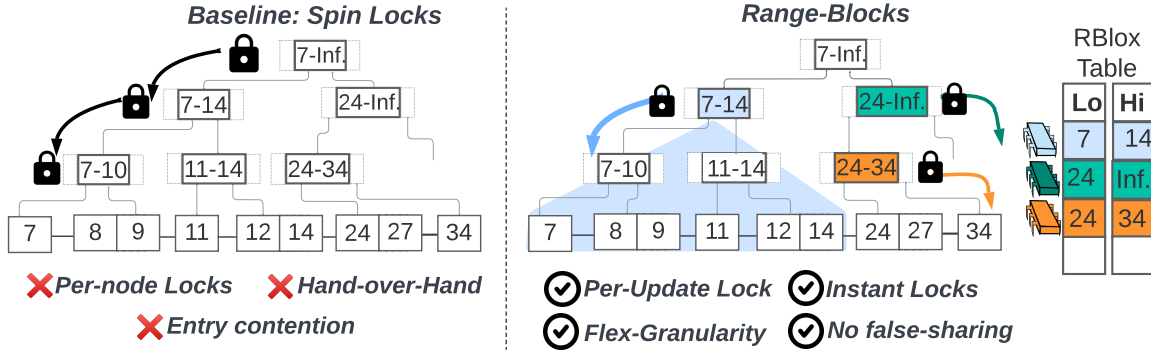


Figure 1.1: Address-based Locks vs. Range-blocks synchronization for a B+tree traversal. For clarity, we use an explicit notation for ranges: Lo (leftmost leaf) and Hi (rightmost leaf) leaf. Inf: Max leaf value in the tree.

1.1 Mutations on host

Currently, DSAs operate solely on read-only workloads, with interspersed minor updates (which do not structurally modify the underlying data structure, e.g., UPDATE WHERE). Mutations to the data structure are pushed to the host machine, which performs data structure inserts using mutexes and atomics, stalling the readers on the DSA. After the inserts are done, the DSA continues with the reads. DSAs do not perform the inserts because they currently do not support atomics or LL-SC since DSAs do not do loads and stores but perform access-execute cycles.

Let us take an example of a dataflow accelerator that uses B+Trees as an underlying data structure [60] for a simple database scan workload; here, the reads happen on the DSA but on encountering an INSERT INTO statement, the readers are stalled and updates to the B+Tree are pushed to the host which performs mutations, synchronized using address locks (figure 1.1). A flag is associated with each node, which we grab and release as we traverse to permit multiple threads to enter. The term address locks refers to the notion that locks are associated with individual nodes [10] and are not explicitly aware of the data-structure hierarchy. This is a consequence of the locks being implemented using hardware atomics that work on memory addresses, i.e., Load-Linked Store-Conditional (LL-SC) and Compare-and-Swap (CAS). Consequently, multiple challenges arise: i) **Address locks introduce high penalty.** We find that even with hardware acceleration (e.g., lock cache [67, 65]), the performance penalty is prohibitive due to the number of locks. Address locks must contend with the uneasy tradeoff between lock granularity and parallelism. ii) **Per-node locks, lead to a large sync working set.** Mutations will grab locks on each node during a traversal. This leads to a large synchronization working set and thrashes hardware structures such as lock caches [67, 65]. iii) **Address-based atomics means locks are forced to perform hand-over-hand.** The locks are associated with the node’s address, and in a B+tree, only

the parent includes a reference to the child pointer. Thus, to access each node’s lock, we need to access the parent. This leads to long synchronization chains kick-starting from the root and causes an entrance bottleneck.

1.2 Range-Blocks overview

Our Approach Range-blocks (*RBlox*) introduces hardware support for synchronizing dynamic data structures in DSAs. The central idea is to use the key ranges to demarcate synchronization boundaries. *RBlox* coordinates concurrent operations in the same logical space as that used to organize the data structure and takes advantage of underlying parallelism in the layout. We added a small hardware table where threads register and check ranges to avoid conflicts (Figure 1.1: Right). We observe multiple benefits for range locks compared to address locks: i) **Per-update locks lead to a small table.** Ranges decouple the locks from the nodes and associate them with the operation (or thread). Thus, the number of entries required is proportional to the number of updates. ii) **Range locks compactly represent synchronization boundaries, meaning fewer locks are required per update.** Range locks are symbolic and enable a compressed representation. A range lock on [7–14] serves as a mutex on children 7, 8–9, 11 and 12–14; no need for individual locks. iii) **Instant locks for quick synchronization.** Range locks explicitly identify all locked ranges, which, by process of elimination, also identifies the available ranges. Thus, we can identify and grab unlocked ranges, bypassing the need for ordered traversals, e.g., to insert a key 25, the update has grabbed [24–34] without traversing through ancestors. iv) **Adaptable concurrency with flexible lock granularity** The system can implement coarse-grained or fine-grained locking by choosing broader or narrower ranges (e.g., [7–14] vs. [7–10]). This avoids algorithm redesigns.

1.3 Contributions

Contribution 1 We incorporate *RBlox* into spatial dataflow architectures [59, 60], and enable parallel updates on the DSA. We port five different algorithms: database scans, data analytics [59], pagerank [36], key-value stores [17], OS virtual-memory code [31], and study multiple data-structures: B+trees, sorted-sets, hash-tables, and adjacency lists. Our contributions are as follows:

- We propose a novel hardware facility for synchronization in DSAs based on data-structure key ranges. *RBlox* enables DSAs to work with dynamic data structures.
- We evaluate two variants: *RBlox*, which employs a small table for mutual exclusion only, and *RBlox++*, which enables instant locking and shortens critical path.

- *RBlox* improves performance by $7.8\times$ by streamlining locks. We reduced sync ops by $\frac{1}{3}$ and DRAM energy by 35%. We analytically show that we only need 1 range lock/update, and in the worst case, we need only $O(N_{tiles})$. i.e., 128 entries ($\simeq 2\text{KB}$) for 128 tile DSA.
- *RBlox++* improves performance by $15\times$ by instantly grabbing free range locks. It reduces the number of sync ops on the critical path by $\frac{2}{3}$ compared to baseline. It requires an optional 16KB victim table to reuse unlocked ranges.

RBlox makes novel contributions across two themes: synchronization and spatial dataflow architectures. We briefly state them to help contextualize our work. : Contribution 1: We are the first to explore a practical implementation of range locks for in-memory data structures. Prior work explored ranges for database transactions [27], virtual-memory [31], and file-systems [48]. We exploit ranges to reduce the number of locks, enable instant locking, and improve parallelism (compared to address-based locks). Contribution 2: We are the first to support dynamically re-sizeable and mutable data structures in DSAs. The lack of synchronization in DSAs has limited the types of applications, parallel patterns, and data structures they support. We address this problem. Contribution 3: We are the first to develop hardware support for range locks. Prior hardware support for synchronization has focused on address-based atomics (e.g., [67, 65, 58, 63]). We analytically show that range locks are more efficient (§ 4.2) and compare performance.

Contribution 2 *Why is there a chapter 5? Discussing the relationship between chapters 3 and 5.*

I authored another paper before *RBlox*, called *METAL: Caching Multi-level Indexes in Domain-Specific Architectures*. *METAL* is an intelligent caching framework that can be ported to spatial data flow accelerators for speeding up read-only workloads. *METAL* has an index cache (*IX-cache*) that stores data structure layout information (we elaborate on this in Chapter 5) for speeding up data DSA walks and employs reuse patterns [47]. *METAL* with only the index cache (*IX-cache*), is termed "*METAL-IX*", which represents my contributions to the paper.

We incorporate *METAL-IX* into four DSAs: *Gorgon* [59], *Capstan* [52], *Aurochs* [60], and *Widx* [29] and target a set of 8 algorithms. In Chapter 5, we provide an overview of the DSAs we target along with a detailed description. We evaluate index reuse in multiple applications: data analytics, database scans, graph processing, spatial analysis, and sparse matrix algebra. *METAL*'s contributions are as follows:

- 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, index cache(*IX-cache*), that uses index ranges as cache tags. Reorganizing the cache helps short-circuit index walks and improves caching efficiency.
- *METAL* with index cache (termed *METAL-IX*) performs $2.8\times$ better than address-cache, and $1.6\times$ better than X-cache [54] (state-of-the-art DSA cache). We save $1.4\times$ bandwidth vs. address and $1.2\times$ vs. X-cache.

Contribution 3 Relation between my thesis and Aditya Prasanna’s M.Sc. thesis from Simon Fraser University [47]. I worked on *METAL-IX* (chapter 5). Aditya Prasanna’s thesis explores reuse patterns as an optimization to *METAL-IX* (called *METAL*). More details on *METAL* (*METAL-IX* + Reuse patterns) can be found in his thesis.

1.4 Publications

This dissertation includes work published at ASPLOS’24, a peer-reviewed conference. I have collaborated with my supervisor, Dr. Arrvinth Shriraman, to conduct the research discussed. The publications are listed below, along with the contributions of the authors:

- **ASPLOS’24 - METAL: Caching Multi-level Indexes in Domain-Specific Architectures** was previously published at the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, with equal contribution author, Aditya Prasanna and supervisors Dr. Arrvinth Shriraman and Dr. Jonathan Balkind.
 - *Anagha M Anil Kumar*: Contributed to the index cache: idea, design, development, and evaluation of the work. I implemented the *IX-cache* and its interfacing with other modules on Gem5-SALAM [51] (a cycle-accurate hardware accelerator simulation environment), constituting the *METAL-IX*. I also performed the workload setup for *METAL-IX* and Baseline.
 - *Aditya Prasanna*: Contributed to the Reuse Patterns: idea, design, development and evaluation of the work, along with workload set up for *METAL*.
- **ASPLOS’25 - RANGE-BLOCKS: A Synchronizing Facility for Data-Structures in Domain-Specific Architectures** was submitted to the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, with co-author Aditya Prasanna and supervisor Dr. Arrvinth Shriraman. It is currently in **Major Revision**.
 - *Anagha M Anil Kumar*: Contributed to the idea, design, and development of *RBlox*, *RBlox++*, and *RBlox* API. Completed the software implementation

and also simulated on Gem5-Salam [51]. Performed evaluation for *RBlox* and *RBlox++*.

- *Aditya Prasanna*: Contributed to the workload setup and delta pagerank end-to-end implementation. Also performed performance cliff exploration.

1.5 Dissertation Outline

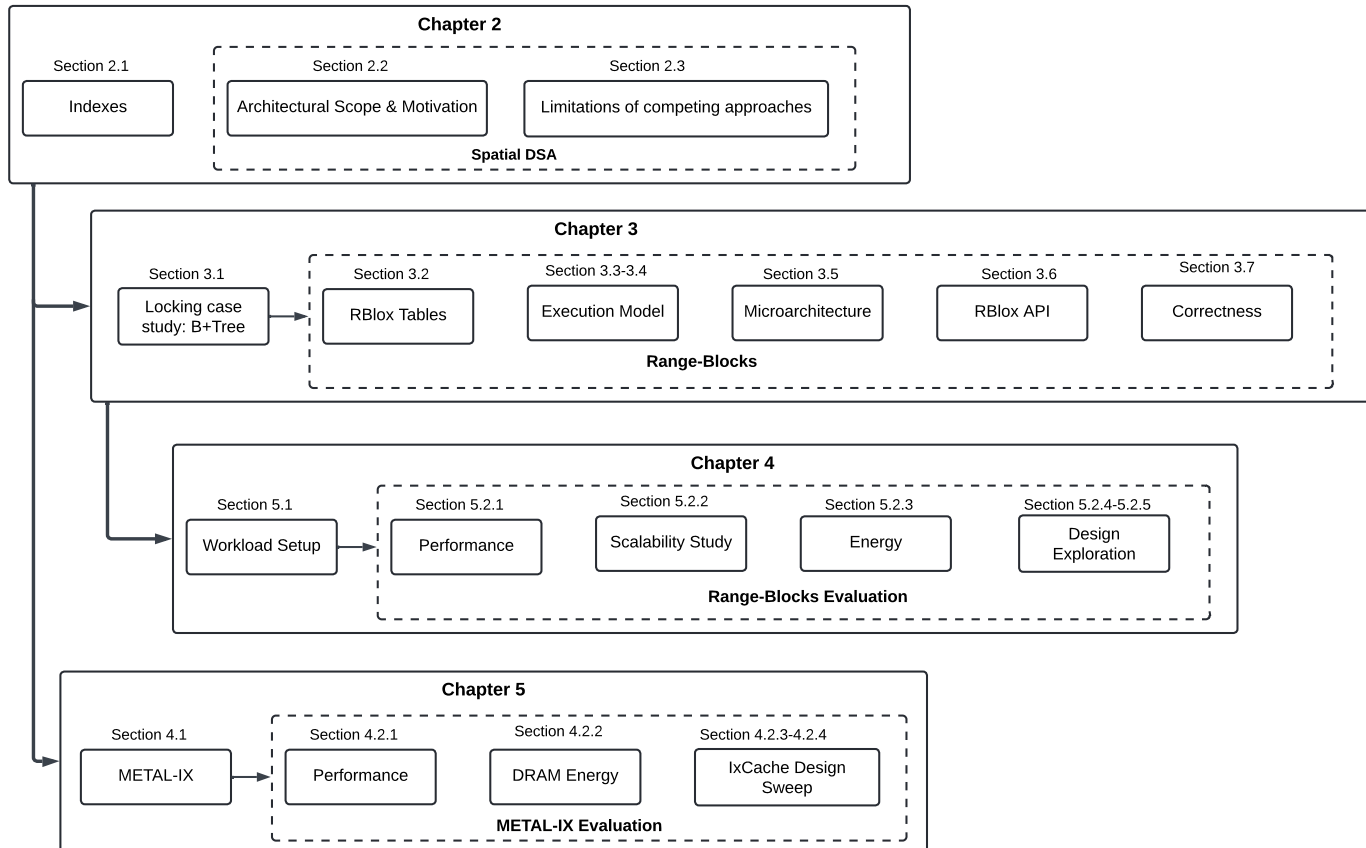


Figure 1.2: Dissertation overview

This dissertation is organized as follows: Chapter 2 describes the related work, scope, and motivation for *RBlox* and *METAL-IX*. Chapter 3 describes *RBlox* and its architecture. Chapter 4 describes the extensive evaluation carried out on *RBlox*. Chapter 5 describes *METAL-IX*, whose *IX-cache* inspired the hardware for the *RBlox* tables. The summary is outlined in chapter 6.

Chapter 2

Impact and Scope

This chapter details the importance of indexes, followed by their challenges and deployment in current state-of-the-art DSAs. We motivate that *METAL* speeds up reads in DSAs by addressing the challenges of *data-structure* walks (§ 2.1). DSAs currently lack a synchronization facility for writers. We motivate and develop *RBlox* to address these challenges (§ 2.2) and discuss the limitations of previous work § 2.3.

2.1 Indexes in target DSAs

Indexed *data-structures* or indexes are data storage formats that organize and manage data efficiently. They are of different types, such as B+Trees, Hash Tables, Graphs, etc. They organize data in a way that significantly reduces search time, enhancing the performance of operations such as querying and updating records, resulting in extensive usage in database systems [40] for query execution speed optimization, as a product of reducing the amount of data scanned. Indexes are also used in search engines to handle large-scale data efficiently. Sparse Matrices employ indexes to represent and query data with low memory consumption, indexes additionally optimize compute by parallel handling and accessing non-zero data elements; Overall, indexes are invaluable to processing large volumes of real-world data and contributing to more responsive systems.

State-of-the-art spatial dataflow architectures support indexes. They provide a streaming interface that streams every node access from memory and a small scratchpad to exploit immediate reuse. However, they do not account for locality reuse or index updates. Indexes are structured to store data in a hierarchical and efficient way. The DSAs walk indexes to access data objects [60, 54], and we find that the walk time accounts for 30%-90% of end-to-end time.

Address-based caches are well understood to capture reuse in indexes and reduce the walk latency by cutting memory access time. However, they have organizational flaws, and we illustrate them as challenges while caching for a classic index - the B+tree. **Challenge 1: Address caches always require root-to-leaf walks, which overflow the cache**

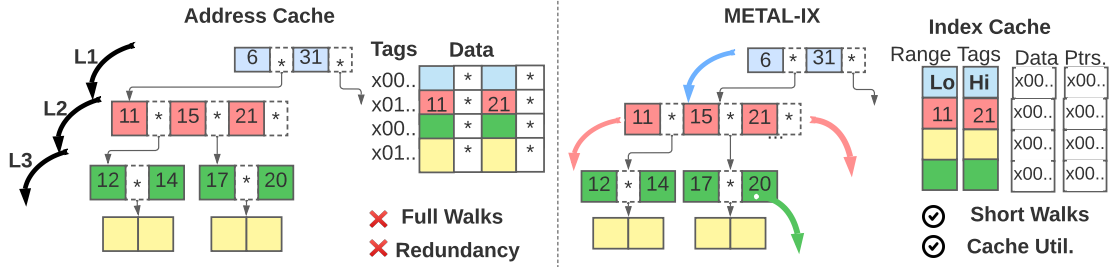


Figure 2.1: index walk with address cache vs. with *IX-cache* (*METAL-IX*).

capacity. The address cache is tagged by addresses. But each node’s address is only accessible from its parent, so walks traverse the whole B+Tree. Even if a lower node is cached, we cannot reach it until the parent is read. A hit or miss in the address cache triggers root-to-leaf walks. However, on a hit, the address cache eliminates 1 memory access. **Challenge 2: Address caches are polluted with redundant nodes.** Address cache caches all nodes touched along a walk to minimize the walk latency. However, this results in redundant entries because if we could identify the level already cached, we wouldn’t need to traverse the redundant levels above it. **Challenge 3: Tradeoff between reach and effectiveness:** Upper nodes are common across multiple walks and maximize the reach while lower nodes effectively short circuit and reduce walk latency - this tradeoff requires knowledge of the workload, which the address cache is blind to since it tags by addresses.

We observe that the same key is searched at all levels to locate the appropriate data object during a walk. Lookup at every internal node can be redundant. This paves the path to the idea of "short-circuiting", that is, traversing an index to retrieve a data object by caching an internal node, reducing the number of redundant comparisons, especially in deep indexes.

This motivated the development of the *IX-cache*, as an integration onto state-of-the-art spatial DSAs to speed up index walks. Caching *data-structure* ranges helped start the traversal from intermediate nodes instead of always starting from the root node, speeding up index walks on the DSA, resulting in faster reads by 5.3×.

The *IX-cache* is a novel cache architecture that inverts the organization of the address cache, and the [Lo–Hi] range in the index node constitutes the tag. The DSA probes the *IX-cache* using index keys and can kickstart the walk from the cached node closer to the leaf on a hit. Interestingly, beyond reducing the walk latency, a primary benefit of short-circuiting is reducing the working set size (the total number of nodes touched during a walk). Per tag matches require more energy than address cache, but fewer accesses mean lower overall energy.

IX-cache can cache varying granularity of ranges - caching for reach would cache a coarser-grained range - to short-circuit multiple walks, and caching for effectiveness would cache a finer-grained range - to short-circuit more effectively and reduce walk latency.

METAL employs reuse patterns in addition to *IX-cache*, which is further discussed in Aditya Prasanna’s M.Sc. thesis (SFU) [47]. We term *METAL* with just *IX-cache* and without using reuse patterns as *METAL-IX*.

2.2 Architecture Scope: Spatial Dataflow Architectures.

METAL speeds up reads on DSAs through short-circuiting but does not provide a synchronization facility to speed up writes on DSAs. In fact, this is a gap in current state-of-the-art dataflow architectures. We aim to address the paucity of research on synchronization in domain-specific architectures (DSAs), the platform of choice for many applications [23]. *RBlox* evaluation reveals that our design does well against state-of-the-art synchronization from CPUs and GPUs, lending confidence that the ideas are architecture-independent. The architectural template for many DSAs in industry [45, 53, 1] and academia [39, 64, 16, 29, 52, 60, 15, 59] is spatial dataflow. Spatial dataflow organizes the chip as a distributed mesh of compute and memory tiles, with many independent memory spaces such as scratchpads, hardware FIFOs, and pipeline registers. They eschew caches, coherence, and atomics to maximize compute-memory bandwidth and minimize data-movement energy. Due to the dearth of synchronization, the generality of current DSAs is limited: i) they support either read-only [60] or narrow write behaviour, e.g., reductions [21] ii) they employ space inefficient affine static *data-structures* (e.g., bitmaps) [26] that cannot handle data resizing. iii) they only target deterministic algorithms [30] and cannot support faster-converging dynamic algorithms [7].

Incorporating mutating *data-structures* in spatial dataflow pipelines is challenging: i) *data-structure* writes require support for generalized multiple reader-writer synchronization, while spatial dataflow is limited to single producer-single consumer. ii) Both the mutating threads that need to be synchronized and when they need to be synchronized are only known at runtime. Unfortunately, in spatial dataflow, all thread interactions must be statically analyzable by the compiler. iii) Finally, in current DSAs, the compiler organizes the data movement between synchronizing threads. However, mutations may dynamically update different parts of the *data-structure* and cannot be statically scheduled.

GPPs use synchronization mechanisms like CAS and LL-SC. CAS is associated per node. Address locks are associated with node addresses. We can put these addresses in a hash table. If we could associate with keys instead of addresses in the table, we would create a new synchronization mechanism through which we can shorten the critical path and also hold fewer locks on the critical path. Usually, more focus is given to the synchronization strategy as the synchronization mechanisms are well-defined in GPPs. Current spatial dataflow architectures do not support CAS and LL-SC (load-linked, store-conditional) since they do not support loads and stores and only operate with access-execute cycles. For DSAs, we have implemented a novel synchronization mechanism - the *RBlox* tables. This table is tagged

by index key ranges. If we hypothetically assume CAS was supported, we could consider synchronization strategies which enable lock-free index implementation [61, 55]. However, *RBlox* tables provide performance gain through instant locking and utilizing 1 entry/update and can change the granularity of locking by grabbing a range higher up or lower down in the tree. These tables are tagged by *data-structure* ranges and hold information on locks, allowing us to support the following synchronization strategies: (i) storing 1 entry/update in the table, reducing the number of synchronization operations on critical paths. (ii) instant locking writes, enabling a smaller working set size and a smaller critical path. We also explore other locking mechanisms in this dissertation, such as L-Cache and R-List, to compare against *RBlox* tables. L-Cache uses hand-over-hand as the synchronization mechanism, and R-List uses range locks.

2.2.1 Enabling Associative *data-structures* in DSAs

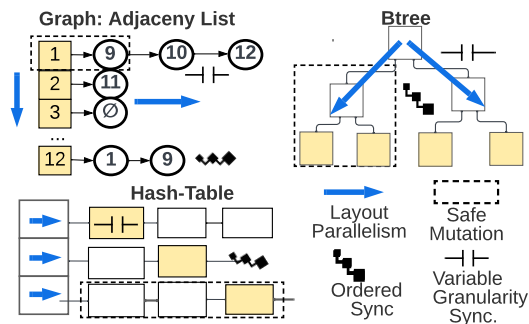


Figure 2.2: Traits of Associative Containers.

Keys and ranges are commonly found in *data-structures* such as dictionaries, hash maps, sorted sets, adjacency graphs, and search trees [20, 50].¹ We highlight the traits that make associative containers ideally suited for handling concurrent mutations and the implications that make range locks a natural fit. i) **Trait: Parallel Layout. Implication: Parallel range Locks** Associative containers, by design, have parallelism-friendly layouts that group keys and traverse them independently, e.g., index nodes in B+Trees, shards in graph lists, and buckets in hash tables. Range locks capture this inherent concurrency in data layout. ii) **Trait: Safe and Sharded Mutations. Implication: Fine-grain range locks** In associative containers, the layout is typically sharded, and mutations are confined to “safe” regions, e.g., within a sub-branch of B+Tree, partition of a graph, bucket in hash-table. This enables narrow range-locks and naturally avoids coarse-grain blocking of threads. iii) **Trait: Hierarchical Synchronization. Implication: Simplified correctness rules** Associative containers are lexicographically ordered, and mutations proceed in an ordered manner.

¹Range locks can support affine *data-structures* with implicit keys (e.g., arrays), but mutations are inefficient and thus not our focus.

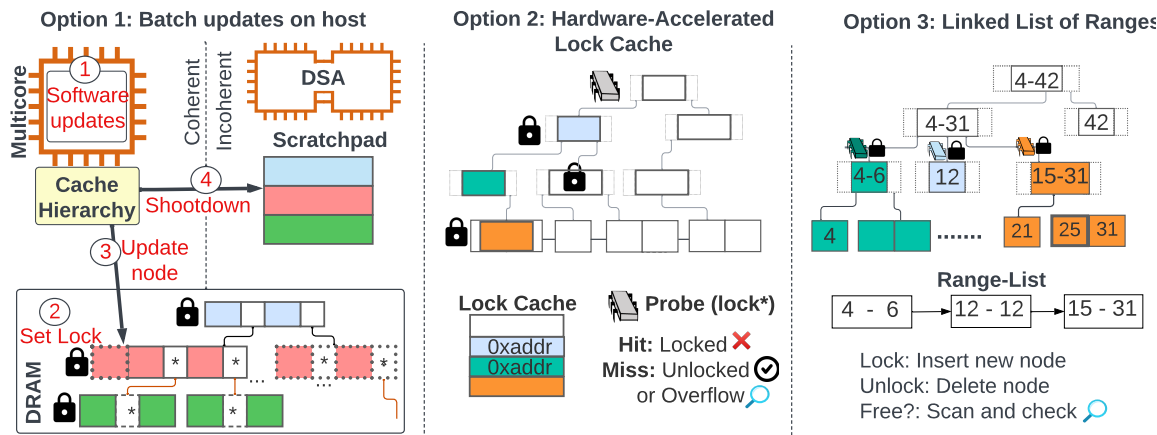


Figure 2.12: Prior State-of-the-Art Approaches to Synchronizing *dynamic data-structures*.

This ensures deadlock freedom for ranges. Data can be inserted or deleted by updating key ranges and redirecting pointers without moving actual data. iv) **Trait: Sparse Nodes.** **Implication: Variable-granularity range locks.** Each region includes a different number of leaf data and is sparse, i.e., it may not include all keys in the region’s sub-range. Thus, a range lock protects a variable granularity of objects and can adapt to contention.

2.3 Limitations of Competing Approaches

There has not been much work on either i) supporting concurrent *dynamic data-structures* in DSAs or ii) analyzing the tradeoffs between address-based and range-based locks for in-memory *data-structures*. In this section, we construct three competing approaches based on prior SOTA in CPUs/GPUs. Here, we qualitatively analyze their limitations (and quantitatively evaluate them in § 4.2.1).

Option 1 (Baseline Hybrid): Batch mutations on the host.

This approach tethers the DSA to a host multicore and then completes the updates [5, 44, 55] on the host in a bulk synchronous manner. The multicore is cache coherent and supports atomics, but the DSA is not cache coherent with the host (it connects through main memory).

We use the lock flag within each node to explicitly synchronize with the DSA (Figure 2.12). During an update, the host sets the flag and blocks the DSA’s walkers from reading the node into the scratchpad. On completing the mutation, it resets the flag and releases the walkers. **Limitations:** i) writer parallelism is limited since the multicore has fewer threads (compared to DSA). We would need a 4–5× larger multicore to improve writer throughput, but the power budget would exceed DSA. ii) each node requires a heavyweight atomic operation, and iii) finally, readers on the DSA are slowed down by writer shutdowns.

Option 2: Hardware-accelerated Address Locks (LCache)

Multiple works have proposed hardware acceleration for address-based locks [65, 67, 33, 19, 58]. These enable *data-structure* mutations to run on the DSA itself and improve the update throughput since DSAs typically include more hardware threads. The idea (Figure 2.12) is to maintain a cache for the locks actively held. To acquire a lock, a thread inserts the lock address into the cache. To check if a lock has been taken, we only need to check the cache for a hit. Lock operations are faster since they avoid memory accesses in the common case. **Limitation:** The lock working set (i.e., number of locks required) is large in *data-structures*, which trashes the lock cache and increases lock latency. We observe that the lock working set is large because of address-based locks.

Option 3: Linked List of Ranges (R-List)

In databases and OS, range locks are maintained in an auxiliary linked list [27, 31]. Range locks are derived from the logical space of the data structure and are physically locked for access. These differ from transaction locks since the former grabs latches at the data structure (lower) level, and the latter operates on the application level. Each node in the list includes the range bounds, the identity of the holder (e.g., thread ID), and additional metadata like lock type (shared or exclusive). The thread inserts a new node into the list to acquire a lock. The nodes in the list are sorted in ascending order based on the *Lo* of each range. The sorted list enables us to check and verify no conflicting ranges (without scanning the entire list). **Limitation:** The primary challenge with R-List is the overhead of verifying a range is available. This requires a scan through the linked list and a check at each node. Sorting helps avoid scanning. Previous literature has not mentioned overheads because they make them untenable for use in in-memory *data-structures*, whereas *RBlox* makes these locks practical.

Chapter 3

Range-Blocks

We perform a case study on the B+Tree locking mechanisms currently used (§ 3.1), followed by our insights and observations on how we can port them to DSAs. We discuss the hardware required on the DSA for the synchronization facility (§ 3.1) and our execution models (§ 3.3). We understand the table microarchitecture (§ 3.5) and the *RBlox* API (§ 3.6). We then discuss *RBlox* invariants and correctness (§ 3.7).

3.1 Locking Fundamentals: A B+Tree Example

Figure 3.1 illustrates state-of-the-art optimistic synchronization [10] in a classic associative container, the B+Tree. Each node contains a lock word, acting as a counting semaphore for multiple readers. During a lookup, the reader grabs and releases each node while traversing the tree, needing only to read a snapshot safely. However, inserts may modify multiple nodes and thus require multiple locks, potentially for an entire branch. These locks must be managed carefully to prevent cascading. In a B+tree, modifications propagate from the leaf upwards. An insert can cause a leaf to overflow, necessitating the expansion of its parent, potentially cascading further up the tree until a node with sufficient space (a "safe" node) is found. Locks must be conservatively acquired in order from the top down during traversal to avoid deadlocks. When reaching a "safe" node, we establish a safe zone and release all

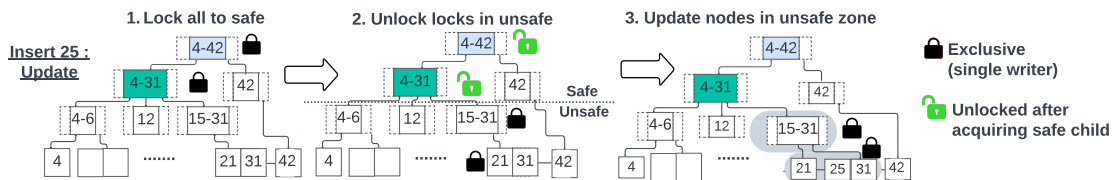


Figure 3.1: Illustration of synchronization in a B+tree. [15–31] is assumed to have sufficient space for expansion. For clarity, we use an explicit notation for each node and list the Lo (leftmost leaf) and Hi (rightmost leaf) keys across sub-branches.

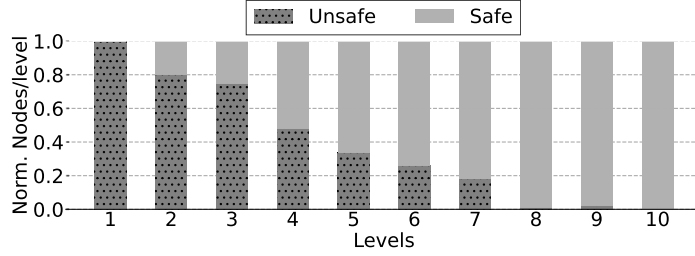


Figure 3.2: Percentage of safe nodes in a B+tree by level. B+Tree depth: 10. # size: 10M nodes. Degree: 5

locks from the root to the safe node. All nodes in the "unsafe" zone remain locked to ensure safety.

Insights The only locks the mutations need are those in the unsafe zone (below a safe node). However, address locks require ordering and hand-over-hand. Therefore, we seek to understand which locks are superfluous by plotting the distribution of safe nodes in a *data-structure* (Figure 3.2). **Insight 1:** Most locks during hand-over-hand are superfluous, i.e., they are immediately released since child nodes are safe (at a lower level). **Insight 2:** In particular, entrance locks at the top level are unnecessary. Safe nodes are frequently encountered lower in the tree, where node ranges are narrower and affect fewer objects. **Insight 3:** There is a potential to expose more parallelism if we have a mechanism to grab the locks of the lower safe nodes. With our idea, if we use the same namespace for synchronization as that used to organize the *data-structure*, we can minimize the region being locked.

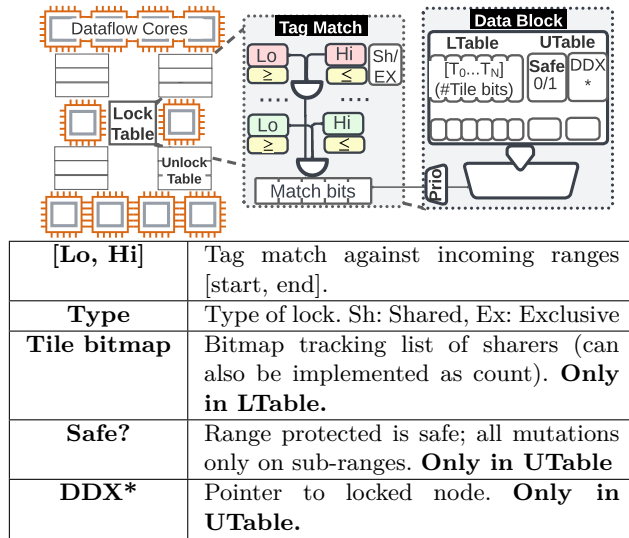


Figure 3.3: Range block table and fields.

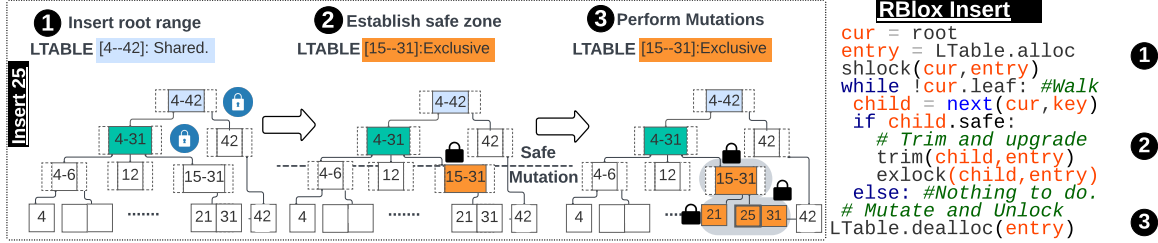


Figure 3.6: Illustration of *RBlox*. Inserts allocate ranges in the LTable; reads self-validate. [15-31] is assumed as safe.

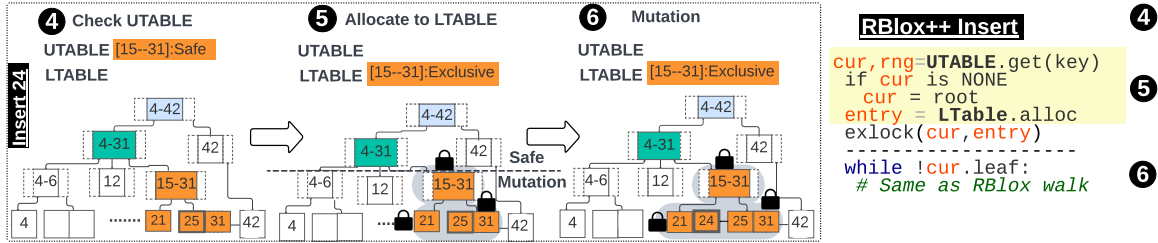


Figure 3.7: Illustration of *RBlox++*. Inserts use unlocked safe ranges in UTable to instantly grab locks to establish a mutation zone.

3.2 Overview of *RBlox* Tables

RBlox hardware design includes two distinct tables to manage synchronization ranges (Figure 3.3). i) **LTable**: This table is essential for tracking the locked ranges actively participating in synchronization. Each concurrent update requires only 1 entry in the table, and the size of the table is $O(N_{parallel-updates})$ ($\simeq 10$ s even on a DSA). ii) **UTable** This table monitors the ranges that are unlocked and often reused. It serves as a temporary holding area for entries that are removed from the LTable, facilitating the instant acquisition of locks to boost performance.

The separation of the LTable and UTable enables a clear distinction between system correctness (handled by the LTable) and performance optimization (facilitated by the UTable).

Let's look at the table organization. The two tables have the same logic for tags, but the data fields are tailored to their roles. The LTable records that the compute tiles are either sharing (SH) or owning (EX) a lock range. It also tracks the active readers using a bitmap to know when to free the entry (a count would suffice, but bitmaps are more energy-efficient). The UTable keeps track of recently used but unlocked ranges. This includes the address of the corresponding *data-structure* node or region and whether it is safe (can accommodate a mutation). Each entry in both tables is tagged with the [Lo, Hi] tuple, indicating a specific range of keys. To ensure no conflicts, we check the tags before adding a new range in the tables. In the upcoming section, we describe the tag matching in more detail.

The UTable and LTable are tagged like a cache, but they do not handle misses and evictions like a cache. Misses are handled by the thread and do not require hardware action. A miss in the LTable indicates a free range lock, and a miss in the UTable means fast-path locking is not feasible and falls back to ordered locking (we discuss this in the next section). On evictions, the UTable can silently drop entries without affecting correctness. LTable never encounters evictions since it only needs to accommodate 1 entry per hardware thread, and we can size it to the hardware concurrency.

3.3 Execution Models

Range-Blocks contributes 2 execution models - *RBlox* and *RBlox++*. *RBlox* implements a streamlined version of hand-over-hand locking, and only needs the LTable. *RBlox++* uses the UTable to implement instant locking i.e., we avoid ordered traversals of the *data-structure* and directly acquire the region.

***RBlox* – Hand-over-Hand using LTable** Figure 3.6 illustrates an insert. ❶ Initially, a single entry is allocated in the LTable for the root’s range, [4–42]. This entry is set as a shared lock, allowing multiple operations to enter the *data-structure*. We will reuse and trim the entry’s ranges as we approach the target region for modification. ❷ Next, we traverse the *data-structure*. Since the root range compactly encompasses sub-ranges (e.g., [4–31]), no additional steps are needed until we reach a safe node. Upon reaching such a node, we trim the range (adjust the lower and upper bounds) and switch the lock to Exclusive to prepare for the mutation. ❸ Finally, the synchronization is complete and the update can safely mutate the region. In this example, [15–31] is assumed to be safe (and capable of accommodating new entries). A single exclusive lock in [15–31] is all that is needed to carry out all mutations on leaves, [21] and [31] since they fall in the protected range. Finally, upon completion, we remove the entry in the LTable, permitting others to enter the branch.

***RBlox++* – Instant locking of free ranges** Ranges are symbolic and can instantly validate whether there is an overlap with active updates. However, *RBlox* is still hand-over-hand. Consider the insert in Figure 3.6. Once it reaches the safe node [15–31], it trims and releases. Our insight is why not grab the safe range [15–31] instantly if there are no conflicts. To support this, we leverage the UTable. When a range lock is released from the LTable (e.g., [15–31] in Figure 3.6), it is inserted into the UTable along with whether the range is safe, i.e., subsequent writes will be localized (we define and illustrate in detail in the § 3.6). Read-only traversals also fill UTable as they encounter safe nodes. Now consider a new insert in Figure 3.7. ❹ We first check the UTable for unlocked ranges using the key to be inserted. If a match is found, then it implies that there is no active mutation in the region, and we instantly kickstart from there. ❺ To grab the range, we move the entry from

the UTable to the LTable. If there is a racing update, one of them will complete it first and make progress. The rest of the traversal is the same as *RBlox*. This approach frees up the entrance to the *data-structure* and reduces the critical path. If no match is found in the UTable, the operation defaults to beginning from the root. When multiple entries in UTable match, we pick the smallest safe range; if both [4–31] and [15–31] are available, we will pick [15–31] since it is narrower. If only [4–31] is available, we will kickstart from it but trim it as soon as we reach [15–31]. This strategy is effective as there are often many unlocked, safe nodes at the bottom of the tree, leading to notable speedup (see Figure 3.2).

Lock-free reads Read-only operations can be made optimistic and synchronization free [10]). Readers only need to validate the pointers during the traversal of the *data-structure*, and they can do so without holding any locks (Figure 3.8). We embed a version number within each node (in memory). When traversing the *data-structure* the readers check the version number before and after using the node’s contents to ensure an intervening write has not modified it. When a writer physically changes a node (e.g., updates key ranges or adds a new branch), it increments the version number. Deletions could prove to be dangerous. Thus, we simply mark the node and lazily clean it up as is convention [24]. **Progress** On an intervening conflicting write, the reader restarts from the root. If starved, the reader will insert the root range into the LTable. This gets trimmed as it makes progress, allowing writers to enter but only after the reader.

***RBlox* vs Transaction Locks** *RBlox* differs from transaction locks on multiple fronts. To summarise:

- **Lock operations:** A single lock operation is enough for a transactional lock. In contrast, for *RBlox*, the number of operations may vary, but it only requires a single entry in the LTable.
- **Nested Locks:** A transaction may have multiple nested locks that need to be released in order. *RBlox*, on the other hand, upgrades locks to finer-grained within the same table entry.
- **Lock escalation:** Transactional locks increase coarseness to reduce overhead instead of having multiple finer-grained locks. *RBlox* increases the fineness of locks to increase concurrency, instead of intent locks higher up.
- **Ordering:** Transactional locks require explicit ordering to guarantee progress and have deadlock freedom. *RBlox* guarantees progress and deadlock freedom. A lock is grabbed on a range in the tree, and traversal is top-down. Therefore, as the walk progresses, we only contract the range. More details on this in section § 3.7

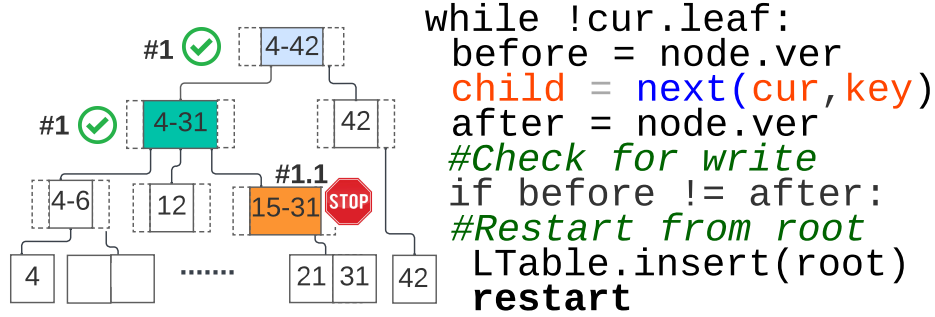


Figure 3.8: Illustration of lock-free reads.

3.4 Why range locks are more efficient in hardware?

Table 3.1 summarizes the discussion. **1. Number of entries per update. Ans: 1 vs. $O(N_{nodes-per-walk})$ for address locks.** Range locks are symbolic and compactly protect nested objects without requiring separate locks on each. Further, hand-over-hand can be implemented by trimming. In spin locks, we would have as many locks per update as nodes in the branch. **2. Total lock working set. Avg: $O(N_{concurrent-updates})$ Worst: $O(N_{tiles})$.** Since each update requires 1 entry, the number of entries required is the number of concurrent updates. On a DSA, the number of concurrent updates in the worst case equals the number of hardware threads. Thus, the LTable can be sufficiently provisioned (and requires no overflow handling). **3. Number of lock probes per update? Ans: $O(N_{safe-per-walk})$ vs. $O(N_{nodes-per-walk})$ for prior work.** Since range locks are symbolic, they implicitly protect sub-ranges. We only probe the table when changing lock type or inserting a new range (see Figure 3.6). Address locks need to grab-and-release at each node. **4. Fully Decoupled Design.** We have separated the maintenance of the ranges for correctness (LTable) and performance (UTable). Each table is independently organized: a monolithic LTable is sufficient since it only contains 10s of entries. UTable is partitioned into banks and sets.

Table 3.1: *RBlox* vs Lock Caches.

	Address-based Locks	<i>RBlox</i>
Application	General (locks, barriers, conditional)	Dynamic <i>data-structures</i>
Target HW	Multicore/NDP (10s cores)	Tile Dataflow (100s PEs)
Lock Type	Physical; address-based	Symbolic (<i>data-structure</i> keys)
Locks/Update	$O(N_{nodes-per-walk})$. Grab-and-release locks at each node.	1 entry/update. Reuse entry to trim range
#Total Locks	$O(N_{nodes-per-walk} * N_{updates})$; Worst: $O(N_{total-nodes})$	$O(N_{updates})$ Worst: $O(N_{tiles})$
#Lock Probes	$O(N_{nodes-per-branch})$	$O(N_{safe-nodes/branch})$
Overflows	Yes #locks > #threads	No #locks = #threads

3.5 Table Microarchitecture

We now describe the microarchitecture and physical implementation of the LTable and UTable. Both tables employ the same tag logic and differ only in the data fields. Each entry in the table represents the exact [Lo, Hi] range lock requested by the application, i.e., the application never encounters false-sharing. The table supports the following range checks:

- i) for a given key (k), the tables verify if $k \notin [\text{Lo}, \text{Hi}]$, i.e., $k < \text{Lo}$ or $k > \text{Hi}$. An LTable non-match indicates the key is unlocked; UTable non-match simply means fall-back to *RBlox*.
- ii) for a new range [start, end], we check $\not\propto [\text{Lo}, \text{Hi}]$ for all entries in LTABLE. If there is no overlap with any entry, then the lock is available. Since $\text{start} < \text{end}$ and $\text{Lo} < \text{Hi}$, we can check a non-overlap with two comparisons - $\text{end} < \text{Lo}$ or $\text{start} > \text{Hi}$.

For hardware scalability, we organize the table into banks and sets. To calculate the bank and set IDs, we segment the ranges; however, each entry itself is maintained precisely. On an insertion, if a lock's range falls entirely within an aligned segment, then it maps to a single bank and set. If a lock range crosses an alignment boundary, we split it into multiple sub-ranges. Each sub-range will fit within an aligned segment and is independently allocated to its bank and set. Any bank and set mapping function is feasible. For instance, here, the range [1–12] is divided into [1–7] (fits within aligned segment [0-7]) and [8–12] (fits within aligned segment [8-15]). The sub-range [1–7] has been mapped to bank 0 and [8–12] to bank 1 (Figure 3.9).

Now consider a probe [start, end] of the table; we need to consider all possible entries that may include any of the keys in [start, end]. The common case is when the [start, end] falls in an aligned segment and does not cross boundaries. Here, we only need to probe a single entry in a unique bank and set. For example, consider that for any of the ranges [1-6],[2-4],[3-7], we only need to check the bank 0, set 0. If a [start, end] check crosses segment boundaries, we split the range along the alignment boundary and check each bank independently – our segment mapping still narrows the number of entries searched. If any check is true, this indicates some partial conflict, and the lock is denied. For instance, a check for the range [4–9] is divided into [4–7] and [8–9], each segment checks their banks in

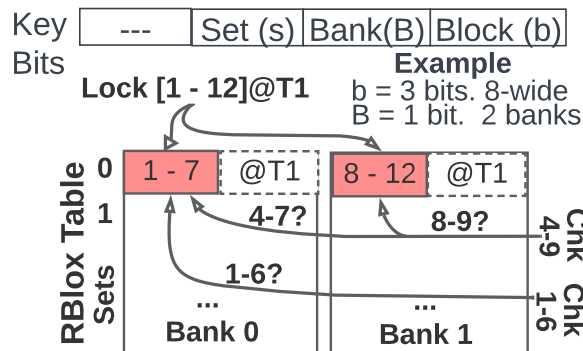
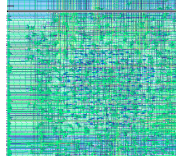


Figure 3.9: Organizing *RBlox* table into banks and sets



Comparator Logic.						
Ref.	nm	Vdd	Trans	Bit.	mW	ns
[57, 12]	180	1.8	800	64	0.7	0.5
[43]	90	1.0	1051	64	1	0.23
[9]	90	1.2	–	64	0.9	0.85
[18]	90	1.0	1359	64	0.8	0.22
<i>RBlox</i>	45	0.85	1400	2×32	0.02	1

Figure 3.10: Right: Synthesis of tag match logic in Nangate 45nm. depth = 10, entries = 256. Left: Floorplan

parallel, and then we combine them. We find that range checks that span multiple banks are uncommon since *data-structures* optimize lookups to narrow down ranges. We do model the parallel lookups in our evaluation.

We also implement the RTL for LTable and UTable in Chisel, synthesize it (Figure 3.10), and compare our complexity against prior literature (Fig. 3.10). Other tag implementations are possible [57]. As an upper-bound estimate, we synthesize using nangate 45nm PDK and OpenROAD [4].

3.6 Range Lock API

Range locks are designed to be a drop-in replacement for address locks in *data-structures*. We first discuss the API and then illustrate usage. Shown below is the lock acquire.

```
bool r_lock(uint Lo, uint Hi, bool type, uint lt_idx):
```

- **[Lo,Hi]** specify the bounds of the range lock.
- **type** can be shared (1) or exclusive(0).
- **lt_idx** is entry in the LTABLE allocated for the lock.

A few design details: i) **How do we identify [Lo, Hi]**? Ranges are defined within the *data-structure* nodes and available during the traversal, e.g., internal nodes (B+trees), skip-nodes (skip lists), vertex lists (adjacency graphs). ii) **What about non-integer key ranges, e.g., strings?** We hash all key types to an integer key, similar to surrogate keys in databases; any hash function will do. Range locks will work as is, conservatively serializing objects with the same hash. However, the goal of a typical hash function is to avoid collisions, which will also avoid overlapping range locks. This approach cannot support range queries as we are lenient on maintaining ordering on hashing and do not support ordered hash tables. iii) **How do we handle contention?** First, range locks support all the standard strategies (e.g., ticket, queueing). They are independent of the lock itself and can be maintained as part of the *data-structure* in memory. Note that range locks are instantaneous (see Figure 3.7) and inherently minimize contention. Thus, *RBlox* uses simple back-off.

```
void r_unlock(uint lt_idx, node_ptr = NULL, bool safe).
```

lt_idx specifies the entry to be cleared in the LTable. **node_ptr** and **safe** are optional parameters for accelerating future lock requests.

The default unlock simply clears the LTable entry. When **safe** and **node_ptr** are provided in unlock, we register the range in the UTable. This permits a fast-path lock that can instantaneously lock the range without having to traverse *data-structure* nodes to obtain the ranges. **node_ptr** is the address of the *data-structure* node associated with the range, e.g., internal index node in B+Tree, skip-node in the list, bucket in the hash table. The concept of **safe** node can be found in all self-balancing *data-structures*. It indicates that a future update to the range will be self-contained and not expand to neighbouring ranges, e.g., updates contained within a hash-table bucket without resizing the bucket array.

We provide a trylock method for accelerated instant locks. The thread specifies a key that it is interested in. The range to be locked is obtained from the UTable, which was left behind by a previous unlock. Since we have omitted the *data-structure* traversal, the trylock has to also return a pointer to the *data-structure* node to the thread. The thread uses this pointer to access the region to perform the mutation. The call is non-blocking; if no entry in the UTable contains the key, the trylock fails and returns NULL; in this case, the thread performs ordered traversals.

```
node_ptr* r_trylock(uint key)
```

key: Check the UTable for a range containing the key.

3.7 *RBlox* Invariants and Correctness

Rule#1 - Mutual independence. The ranges in the LTable (locked) and the UTable (unlocked) are mutually independent, i.e., $\nexists [Lo_{LT}, Hi_{LT}] \ 1.5 \cap [Lo_{UT}, Hi_{UT}]$.

Rule#2 - Locked/Free Any $[Lo-Hi]$ is considered locked if $\exists [Lo_{LT}, Hi_{LT}]$ in the LTable that overlaps; otherwise it is unlocked. UTable is best effort and requires no rules.

Rule #3: LTable Exclusivity: If there exists a table entry for the range $[Lo-Hi]$ exclusively owned by tile T1, there cannot exist exclusive lock owned by T2 for $[Lo'-Hi']$ such that $[Lo'-Hi'] \ 1.5 \cap [Lo-Hi]$.

Rule #4: Lock expansion prohibition: A lock owner is not allowed to expand. If a tile holds $[Lo-Hi]$, it cannot request $[Lo_{new}-Hi_{new}]$ such that $[Lo-Hi] \subset [Lo_{new}-Hi_{new}]$.

Rule #5 Lock Contraction: Only the lock owner is allowed to contract. i.e., if a tile T1 holds $[Lo-Hi]$, only T1 can request a $[Lo_{new}-Hi_{new}]$ such that $[Lo_{new}-Hi_{new}] \subset [Lo-Hi]$.

We illustrate invariants using a visual aid (Figure 3.11).

Invariant #1: Mutual-Exclusion At any given moment, only a single processing unit (referred to as a "tile") can have control over a specific range. According to Rule #3, if a tile T3's requested range $[12-24]$ overlaps with a range $[7-14]$ owned by T1, then T3 will have

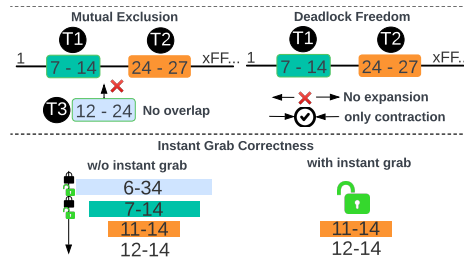


Figure 3.11: *RBlox* Correctness Illustration

to wait. Further, Rule #4 prevents any tile from extending privileges. Here, tile T1 cannot widen the current range [7–14], which guarantees exclusion from existing ranges held by other tiles.

Invariant #2: Preserving write atomicity. By Rule #3, if a tile T1 owns an exclusive lock on [24–27] that means no other tile T2 can read or write a [Lo’–Hi’] that $1.5 \cap$ or \subset [24–27].

Invariant #3: Progress guarantee We ensure progress by structuring the range requests hierarchically. Rule #3 ensures that once a tile T1 grabs a range [7–14], no other tile can grab a sub-range. Conversely, Rule #5 guarantees that any request from T1 to [Lo–Hi] \in [7–14] will be completed.

Invariant #4: Instant Locking is Equivalent to Ordered Locking. Figure 3.11 compares *RBlox* (ordered locking) vs *RBlox++* (instant locking). We show that both reach equivalent states. In *RBlox*, if we start from the root, we will keep trimming until we reach [12–14], noticing that it is safe and free. In *RBlox++*, we obtain this information from the UTable and reach the same state instantly. Two rules ensure the integrity: Rule #1 ensures that entries in LTable and UTable are mutually exclusive, and Rule #2 ensures that any entry in UTable can be locked without conflicts.

Chapter 4

RBlox Evaluation

This chapter details the workload setup and the evaluation performed on *RBlox*.

4.1 Methodologies

Figure 4.1 shows the system configuration for cycle-validated SALAM simulator [51] from MICRO 2020. Our LLVM compiler maps the computation onto the grid of functional units [62, 51]. We write a functional model of the target DSA in C/C++ and host code to drive it. The DSA definition is lowered using high-level-synthesis to a faithful "execute-in-execute" simulator with timing. We model a 2.5D HBM stack connected to a tiled DSA via an interposer. Our baseline simulator has been independently validated and verified against real accelerators [51]. Existing chips (e.g., [45]) do not include synchronization and are incapable of running our proposed applications. Hence, we need to measure it using simulation. We now describe the applications we ported.

Database Scan [59]. Lock Type - B+tree node

Each compute tile conducts batch updates that are queries of the following type: "UPDATE [fields] SET [values] FROM table WHERE X BETWEEN R1 AND R2". We simultaneously run range scans of the form "SELECT * FROM table WHERE X BETWEEN R1 AND R2". The primary *data-structure* is a B+tree prepopulated with 10M records; updates add more records (max capacity is 4 billion records). We vary the read-to-update ratio from 80:20 to 20:80.

Data Analytics [59]. Lock Type - B+tree nodes and records We model a rideshare application that includes user information and ride histories in a B+tree. We use a set of representative queries from prior work [59]: "UPDATE x SET * WHERE y " and a "SELECT * FROM x WHERE y" in parallel, matching riders to dynamically changing drivers. The fields in each record include the status of the rider, passenger count, and trip length. We initialize with 10M riders.

KV-Store [29]. Lock Type - Skip Point We model a Redis-based sorted set table [50] with 10M keys. The *data-structure* combines a hash table and a skip list per bucket. We

Feature	Config
Spatial Grid	128 tiles. Baseline: 128 tiles @1Ghz. 64k scratchpad/tile. Network: Static mesh. 32b/cycle pipelined.
Compute	32bit. +,× (256), <<,>>,& (256).
<i>RBlox</i>	LTable: 1 entry-per-tile (4 banks). UTable: 4k entries (4 banks, 128 sets). 5 cycles/bank. 2 cycle link. per-bank: 3750 fJ
LCache	LCache: 16k entries (\approx 256KB). 5 cycle (cache only).
Memory	HBM 1000 8 Ch width:128 bit, Bank width: 1024bit. BW:128GB/s, 9pJ/bit [38]. nRAS/nWR 17/8ns
Energy	Reg:50fJ +:210fJ ×:1260fJ Bit:180fJ <<:410fJ.
Multicore	24 cores, ARMv9, 2Ghz, OOO. private L1D=64kB; L2=2MB. Fetch/Issue: 8.wide. HBM DRAM is shared
Design Explore	We investigate 16–256 compute tiles and 256—16k entries UTable.

Figure 4.1: System Configuration

Table 4.1: Evaluation Summary

Question	Answer
<i>RBlox</i> speedup?	$7.8\times$ vs. Base , $5.2\times$ vs LCache, $6.5\times$ over R-List .§ 4.2.1.
<i>RBlox++</i> speedup?	$15\times$ vs. Base, $6.8\times$ vs LCache, $2\times$ vs. <i>RBlox</i> , $12.5\times$ over R-List.§ 4.2.1.
How does <i>RBlox++</i> scale ?	$12\text{--}14\times$ speedup when increasing from 16 to 256 tiles.§4.2.2.
DRAM and on-chip Traffic?	<i>RBlox++</i> saves $4\times$ DRAM and 64% of on-chip traffic vs. LCache.§4.2.3.
LTable and UTable size?	LTable = # of Tiles. UTable*: $\approx 2\text{k--}4\text{k}$ entries.§ 4.2.4.
Energy	6.6% of total on-chip energy. § 4.2.3

Best layout: 4 banks, 128 sets, 8 ways

use a degenerate version of range locks to lock the skip points in the list, and only a single node is required to be locked. The skip point to lock is determined based on key value. Instant grabbing is quite effective, particularly in deep (highly associative) buckets, which potentially improves performance.

Virtual Memory Descriptors(VMA) [31]. Lock Type - VMA regions Range locks have been incorporated into OS kernels to manage access within the allocator of virtual memory area (VMA) regions. One option for the kernel is to use a hash table-like structure, with each bucket organized as a skip list of ranges in sorted order. It uses spin locks to protect individual nodes, which can become a bottleneck. We use range locks instead, and the hash table is a fallback.

4.2 Evaluation

Table 4.1 summarizes the results. We evaluate five systems (see § 2.3): i) **Base**: hybrid model that batches updates on host multicore while readers run on the DSA. ii) **LCache**: includes support for hardware-accelerated address locks in DSA. iii) **R-list**: DSA maintains range locks in a linked list. iv) **RBlox**: DSA maintains range locks in LTable. v) **RBlox++**: DSA maintains ranges in LTable and UTable.

4.2.1 Performance Evaluation

Result: *RBlox* achieves $7.8\times$ speedup over baseline, $5.2\times$ over *LCache*, and $6.5\times$ over *R-List*.

Result: *RBlox++* achieves $15\times$ over baseline, $6.8\times$ over *LCache*, $2\times$ speedup over *RBlox*, and $13\times$ over *R-List*.

We normalize the speedup relative to the hybrid baseline (multicore & DSA) - figure 4.2. *RBlox*'s speedup can be attributed to two factors: i) **Lock Elision**: *RBlox* and *RBlox++* require only 1 entry for each update in the LTable. This entry is updated only when the range is narrowed at a safe node. This streamlines the lock mechanism and reduces the number of synchronization operations on critical paths. SCAN and Analytics benefit from this the most. ii) **Increase in concurrency with instantaneous locks** *RBlox++* grabs instant locks on the narrowest range available in the UTable. This leaves regions open for concurrent access. In particular, many of the ranges left open are wide ranges in the entrance. This allows for many more concurrent readers.

In PageRank, the skip nodes see high reuse, resulting in a high hit rate in the UTable. This leads to two benefits: i) reduction in 60% of sync operations. ii) low lock latency due to immediate acquisition instead of traversal. PageRank, with its high reuse of UTable entries, could function with a smaller 1k entry UTable (§ 4.2.3).

Why competing approaches are slow?

The reason competition is slow is the synchronization mechanism itself (how a lock is grabbed), not the synchronization strategy (which locks are grabbed). *RBlox* has the same synchronization strategy as *LCache* and *R-List*; it mutually excludes the same data-

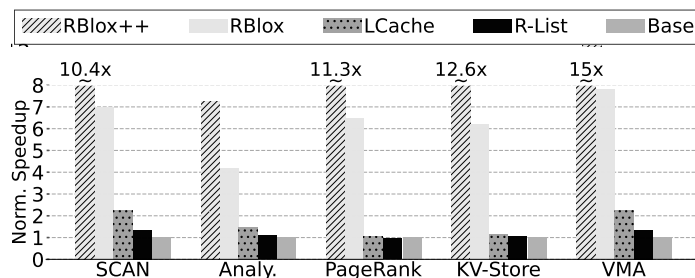


Figure 4.2: Normalized Speedup. Higher is better. Configuration — 20% Updates, DSA:128 tiles. LTable:1 entry/tile. UTable: 4 banks×128 sets×8 ways (4096).

structure regions and has no increase in concurrency. It performs better due to the efficiency of locking. i) The baseline is limited by the number of cores dedicated to writes (24 threads) compared to the DSA (128 threads). If the baseline has to increase writer throughput, it would have to employ more power-hungry general-purpose cores with support for CAS and LL-SC. ii) The LCache has more lock operations relative to *RBlox*. When traversing from root-to-leaf, LCache needs to allocate an entry at each node and trashes the lock cache. iii) R-List uses an auxiliary in-memory *data-structure* (e.g., B+Tree [31]) to maintain the active ranges. This imposes high overhead since lock acquires and releases need to scan the auxiliary *data-structure* iteratively. The main reason for prior approaches’ lower performance is fundamental design flaws (and while the exact numbers may vary), we expect the trend to hold even in alternative cache-coherent platforms. LCache and R-List are state-of-the-art mechanisms in cache-coherent platforms that we have ported to DSAs.

4.2.2 Scalability Study

Result: *RBlox++* improves performance by 11.6–15.3×.

We analyzed the scalability of our system by varying the number of compute tiles from 16 to 128 in *RBlox++* (Figure 4.3). We use three applications to summarize our findings: Scan (high parallelism), KV-Store, and PageRank (moderate parallelism). We maintain a read: insert ratio at 80:20 across all workloads. Both the baseline hybrid and LCache exhibit limited scalability. Their speedup levels off once the core count exceeds 32 to 64. The parallelism of the host multicore constrains the base’s scalability, and LCache’s performance suffers as more tiles trash the lock cache. *RBlox++* consistently outperforms *RBlox* by utilizing instant locking and a more efficient locking strategy. The speedup of *RBlox++* is 15.3×, while *RBlox*’s speedup is limited to 10×. As the number of tiles increases, the reuse of unlocked ranges in the UTable improves. At 16 tiles, we elide 30% of the locks; at 128 tiles, it is 50%. In Scan, *RBlox++* is being limited by the UTable size (fixed across all workloads here) so performance tapers at 128 tiles. If we increase the UTable size, performance improves further(see § 4.2.4).

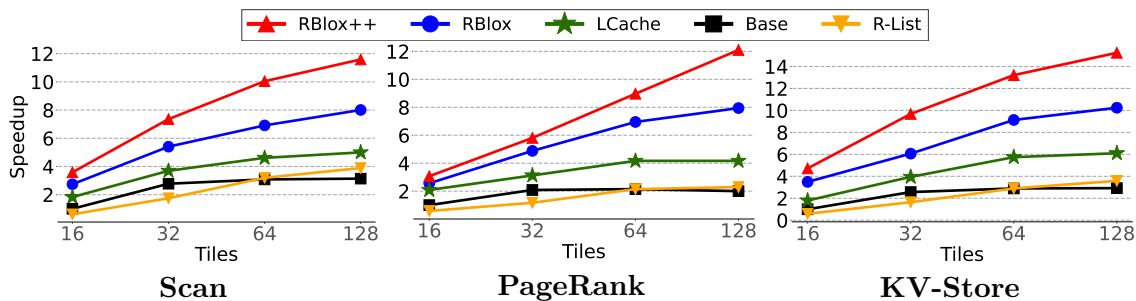


Figure 4.3: Speedup with increasing DSA tiles from 16 to 128. Normalized to 16T, Base.

4.2.3 Energy

Result: Compared to LCache, *RBlox++* improves by $3.9\times$. *RBlox* improves by $1.5\times$.

Result: *RBlox++* reduces on-chip traffic by 70%.

Result: *RBlox* requires only 6.6% of on-chip energy.

We normalize data to LCache since it exhibits the lowest traffic amongst the alternatives (Figure 4.4). R-List wastes significant bandwidth on the auxiliary *data-structure* (40% higher traffic). *RBlox* and *RBlox++* reduce the lock working set by decoupling locks from the *data-structure* nodes and associating them with the update. We only need 1 range lock entry/update, while LCache requires a lock for each node. In *RBlox*, there are no compulsory accesses to the DRAM as all lock operations only involve the on-chip *RBlox* table. Finally, *RBlox++* reduces DRAM traffic further by instant grabbing locks lower in the *data-structure*. This reduces the lock set by $\frac{2}{3}$, and lowers bandwidth.

We also measured the on-chip traffic. Compared to LCache, *RBlox* reduces on-chip traffic by 40%. The main reduction is due to the number of lock operations required. As discussed in § 3.3, *RBlox* needs to update the LTable entry only when trimming ranges at a safe node. However, LCache grabs-and-releases at each node, which necessitates an atomic operation at the lock cache. *RBlox* reduces actual lock operations by $\frac{1}{3}$ compared to LCache. Since *RBlox++* enables instant locking of free ranges, it shorts part of the data-structure traversal and further reduces lock operations by $\sim \frac{2}{3}$ (vs. LCache).

On-chip Energy: Here, we plot the breakdown of the total on-chip energy for *RBlox++* (Figure 4.5) into the following: compute tile, loads and stores, LTable and UTable. Here is a breakdown of the accesses to the tables: i) **UTable:** Both readers and updates access

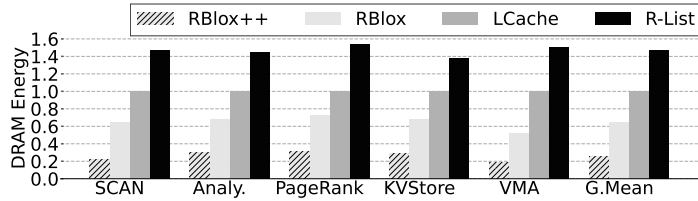


Figure 4.4: DRAM Energy. **Y-axis:** Normalized to LCache.

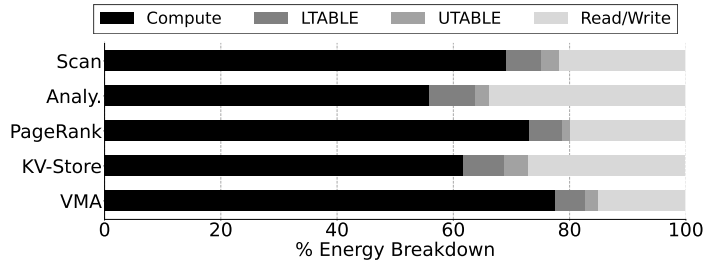


Figure 4.5: Energy Pie. (*RBlox++*) LTable: 128 entries (4 banks \times 32). UTable: 4096 entries. 4banks \times 128sets \times 8 ways.

the UTable 1 time per traversal. This is if there exists a free lock range from where we can start the synchronization. ii) **LTable**: Updates probe the LTable every time they acquire a safe node (see Figure 3.6). Readers need to validate at every node and thus need LTable access. On average, a *data-structure* mutation touches the LTable 4× per traversal (each is 3750fJ). For per-access energy counts, we synthesize the RTL (§ 3.5), and for RAMs, we use CACTI.

4.2.4 UTable size vs. # of tiles in DSA

Result: The best configuration for DSAs up to 256 tiles is 4k entry UTable (organized as 4 banks, 128 sets/bank). Maximum speedup vs. *RBlox* is 5–6×. (Figure 4.6).

We vary the size of the DSA from 32 tiles to 256 tiles to study the scalability. For correctness, the LTable size is always set to the number of tiles. For our baseline experiments in the previous section, we chose 4k entries because it effectively captures the reuse of UTable entries, achieves a high hit rate, and occupies 0.4% of the area of a 128-tile chip. Here, we study UTable sizes from 256 to 16k entries. The UTable is divided into 4 banks and 128 sets/bank (# of ways vary based on size). The red line in the plot is a baseline *RBlox* with 64T (with only LTABLE) to study the impact of UTable.

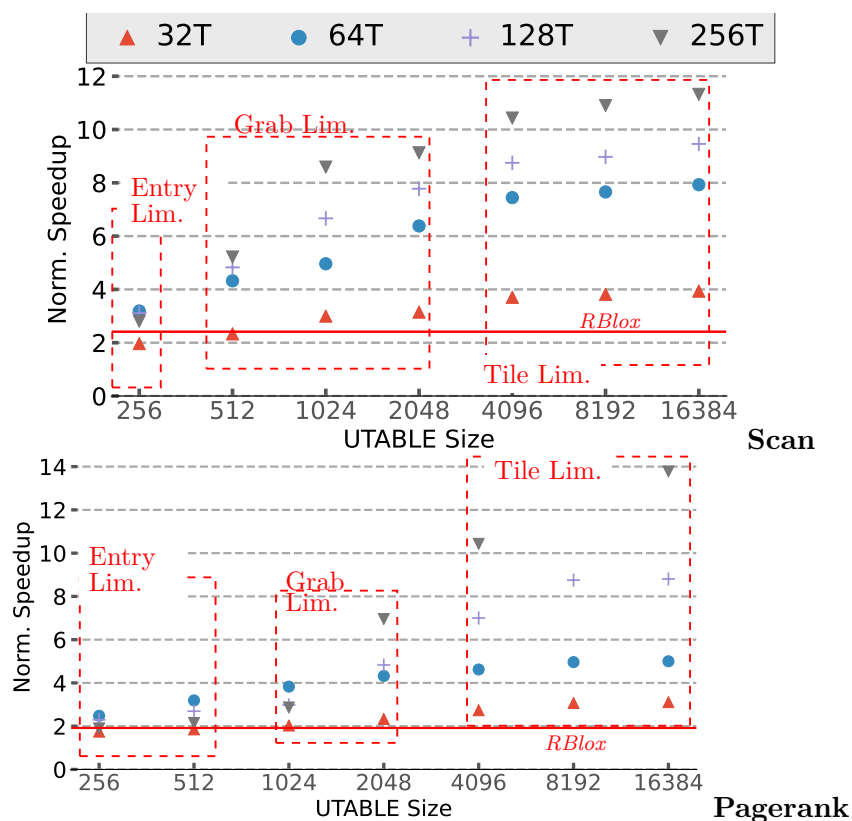


Figure 4.6: Impact of hardware concurrency (#Tiles) vs. UTable size. X-axis: UTable size. Y-axis: Speedup. Baseline: 16T. Red line: *RBlox* 64 Tiles

Our observations: i) *Entries Limited (up to 512 entries)*: Until 512 entries, UTable does not cache sufficient unlocked ranges to aid instant locks. At 128 and 256 tiles, the small UTable is even thrashed, and we find the smaller 64-tile DSA performs better with a 31% higher hit rate in the UTable. ii) *Grab Limited (up to 4096 entries)*: The UTable behaviour is stable and has sufficient capacity to enable instant locking and improve performance. We improve performance by $5\times$ for 128 and 256 tile DSAs by shortening the critical path by 70%. iii) *Compute Limited (≥ 8192 entries)*: The UTable has sufficient capacity; and the workload is compute-limited.

4.2.5 Impact of Update ratio

In Figure 4.7, we study the impact of highly volatile *data-structures* by varying read: update ratios. The trends: i) **For artificially high update ratios ($>40\%$) *RBlox* is competitive. *RBlox++* degrades to *RBlox*** Compared to LCache, both *RBlox* and *RBlox++* scale well since both updates require a single entry (while LCache is thrashed due to number of locks). Ranges enable compact representation of synchronization boundary. *RBlox++*'s speedup does degrade due to the churn in the UTable entries. *RBlox++*'s speedup depends on finding reusable unlocked ranges in the UTable; high volatility changes the unlocked ranges frequently and minimizes this reuse. ii) **For realistic scenarios (update ratio $<40\%$) *RBlox++* provides significant performance boost.** *RBlox++*'s UTable can cache unlocked ranges and instant lock on them. This reduces the critical path by 70% and improves performance by $\simeq 2\times$ compared to *RBlox*. iii) **LCache limited under all scenarios.** Under realistic scenarios (update ratio $<40\%$) LCache is limited by the critical path of an update, multiple locks have to be acquired for each traversal. Under artificially high update ratios, LCache is limited by the number of entries. It is thrashed. It is also limited by the entrance to *data-structure* becoming a bottleneck.

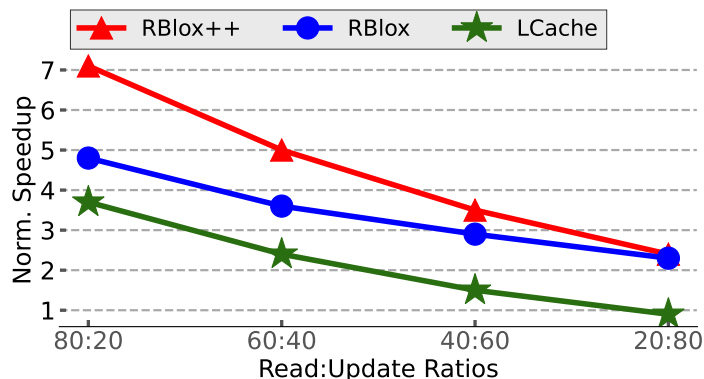


Figure 4.7: Impact of update ratio. LTable: 32, UTable: 4096 (4 banks, 128 sets). Benchmark: Database.

Chapter 5

METAL

This chapter outlines *METAL-IX*, a key contribution of *METAL*, whose *IX-cache* served as the inspiration for the tables in *RBlox*. There are a few differences and similarities between the *IX-cache* and tables, which were highlighted in the *RBlox* table hardware description (§ 3.2). My contribution to *METAL* includes the *METAL-IX* (§ 5.1) and the hard nosed evaluation of the same (§ 5.2).

5.1 Index Cache

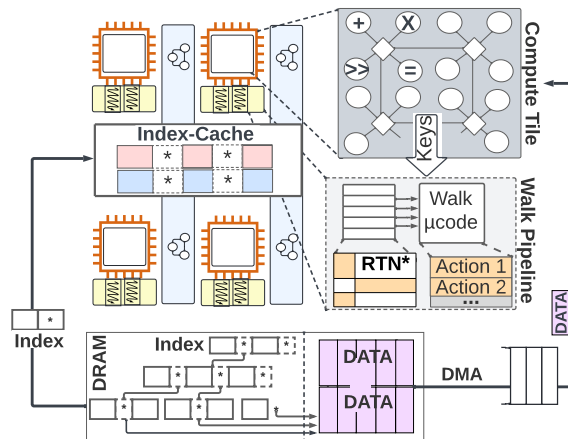


Figure 5.1: *METAL-IX* micro-architecture.

The index cache leverages range tags to short-circuit index walks and reduce the working set and walk latency. It can capture wider index nodes that maximize reach, as well as those closer to the leaf and minimize walk latency, thereby maximizing effectiveness.

METAL incorporates into spatial dataflow architectures [59, 29, 60], which maps the computation onto a grid of compute tiles. Logically, these compute tiles interface with the *data-structure* using keys (not addresses). Each data object in the index is allocated in DRAM and has a unique key that provides a namespace that loads and stores can use. The

index only contains pointers to the data object, and the *IX-cache* only targets the index traversal itself.

METAL adds 2 components - the *IX-cache*, shared by multiple compute tiles to maximize cooperative caching and a pattern controller [47].

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, representing the smallest and largest keys (the range) stored in the block. Fig. 5.2 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].

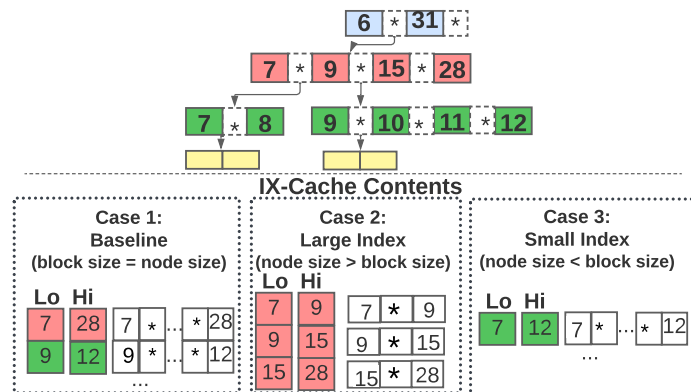


Figure 5.2: Packing index nodes into cache blocks

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 points to the leaf. Every block in *IX-cache* is tagged with the [Lo, Hi] tuple, which represents key ranges encompassed by the block. Fig. 5.3 illustrates the stages in the pipeline (only the first is required, and the 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. 5.3 key 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 ⑩ might match the cache's red and green ranges. 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 10 will match 9-10 in the block [9*10*11]. We achieve this with parallel \leq across all the index keys, then find the 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.

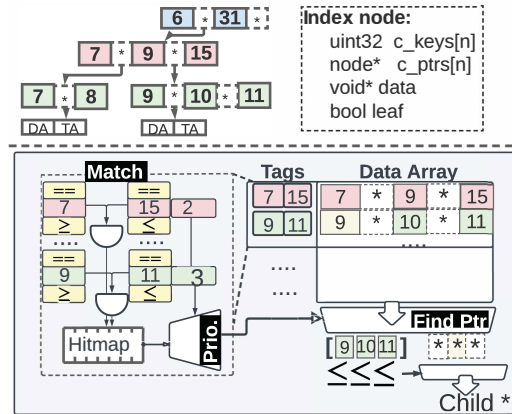


Figure 5.3: Hit Path: *IX-cache* Logic

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 key's LSB. The keys are logically divided into 16 ($b = 4$) wide blocks, e.g., keys 0-to-15 will be a block (Fig 5.4). 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., there is 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 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. 5.4, 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.

5.1.1 Walk Pipeline

Miss Path: A miss triggers a root-to-leaf walk. *METAL* re-purposes the prior microcode engines that the DSAs already include [54, 60]. We only provide an overview due to lack of space. Fig. 5.5 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.

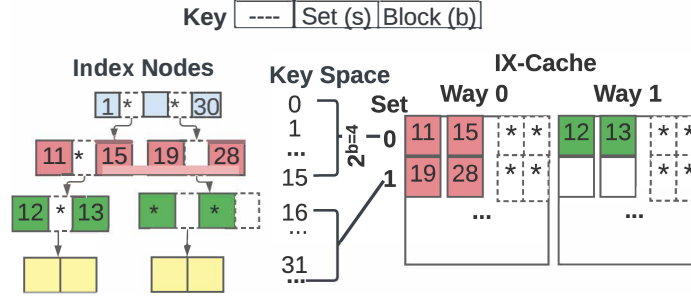


Figure 5.4: Set-associative IX-Cache

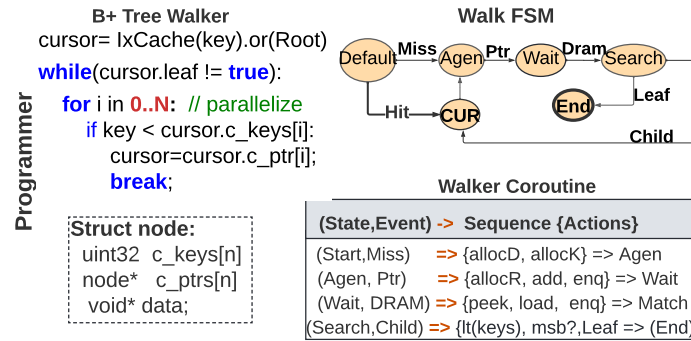


Figure 5.5: Cache miss handler.

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 into a table and microcode.

5.1.2 Similarities and differences between *IX-cache* and *RBlox* tables:

- **MISS:** On a miss, *METAL-IX* initiates DRAM access to retrieve the root node and begins the traversal from there. In contrast, *RBlox* conservatively locks (from the root) to perform writes.
- **HIT:** on a hit, *METAL-IX* short-circuits readers; on a hit, *RBlox* grabs instant lock for writers.
- **UPDATES:** Usually, the entries in the *IX-cache* are only evicted with LRU and not updated. In fact, an entry in the *IX-cache* is not updated during a traversal; *RBlox* has entries in the tables that update often since it deals with mutating *data-structures*. Additionally, during a traversal, an entry can update itself. We use the same entry in the table to store as we traverse down the tree.

- TAG: *IX-cache* and *RBlox* tables use the same tags- *data-structure* ranges.
- DATA: *IX-cache* holds more data, as it stores the entire node, along with a pointer to itself and its children; *RBlox* stores just the pointers to the node and its children. *RBlox* stores lesser metadata.

5.2 METAL-IX Evaluation

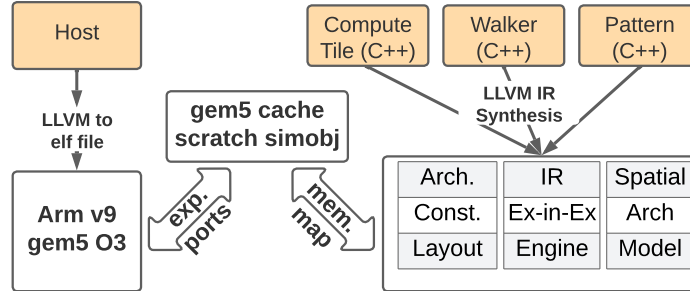


Figure 5.6: *METAL* simulation set up.

Figure 5.6 shows the system configuration for cycle-validated SALAM simulator [51] from MICRO 2020. Our LLVM compiler maps the computation onto the grid of functional units [62, 51]. We write a functional model of the target DSA in C/C++ and host code to drive it. The DSA definition is lowered using high-level-synthesis to 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. All cache blocks are set to 64 bytes to ensure a fair comparison. We deploy *METAL* across 4 DSAs to implement 8 workloads.

We answer the following questions:

- How much can *METAL-IX* improve performance? **Answer:** $5.3 \times$ vs Streaming , $2.8 \times$ vs address based cache and $1.6 \times$ vs X-Cache. § 5.2.1
- How much DRAM energy can *METAL-IX* save? **Answer:** $1.8 \times$ vs Streaming , $1.4 \times$ vs address based cache and $1.2 \times$ vs X-Cache. § 5.2.2
- How much on chip energy is required by the *IX-cache*? **Answer:** 29.5% of total on-chip. upto $5 \times$ lower vs. Addr; $3 \times$ vs X-cache. § 5.2.3
- What is the best configuration for the *IX-cache*? **Answer:** 16-way *IX-cache* § 5.2.4
- Does shared or private *IX-cache* perform better? **Answer:** Walk latency decreases by 6%-95% for private *IX-cache*. § 5.2.5

Feature	Config
DSA Grid	16 (8×2) —128 (8×16) tiles @1Ghz. Statically configured mesh. 32b/cycle pipelined.
Scratchpad Compute Walkers	8K/tile (SpMM) to 64K/tile (JOIN) 32bit. +,× (256), <<, ,& (256). 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

Table 5.1: Overview of Simulation Setup

5.2.1 Performance

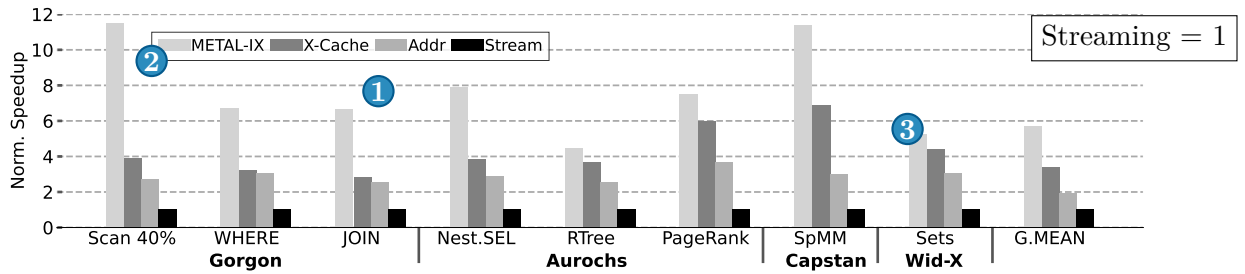


Figure 5.7: Speedup. *METAL-IX* vs. X-Cache vs. Address vs. Stream. (higher is better).

Result: Compared to address-cache, *METAL-IX* short-circuits and reduces working set. Compared to X-Cache, *METAL-IX* allows caching intermediate nodes, not just leaf nodes.

Result: *METAL-IX* performs better by $5.3 \times$ vs Streaming, $2.8 \times$ vs address based cache and $1.6 \times$ vs X-Cache.

Fig. 5.7 plots the speedup. In this section and future sections, the baseline cache sizes are set to 64k, 16-way, and 16 banks. *METAL-IX* achieves speedup by short-circuiting, reducing working set and walk latency. ❶ In workloads with significant working set size - JOIN, *IX-cache* improves by $2.6 \times$ compared to address cache due to short-circuiting and saving multiple DRAM accesses. In contrast, a hit in the address cache only eliminates a single access. We only improve by $2.6 \times$ since JOIN has high arithmetic intensity: 318 ops/walk. We improve performance over X-Cache [54] by $1.6 \times$ overall since we enable caching of intermediate nodes, applying LRU policy to the search batch of keys. ❷ We maximize 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.

5.2.2 DRAM Energy

Result: *METAL-IX* short-circuits and reduces DRAM energy by $1.4 \times$. Compared to X-Cache, *METAL-IX* allows caching intermediate nodes and reduces DRAM energy by $1.2 \times$.

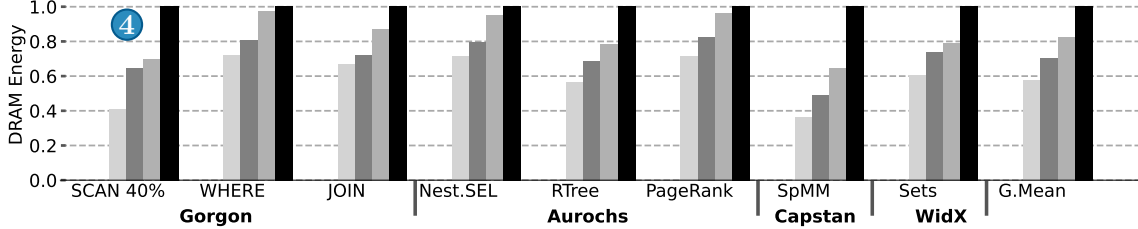


Figure 5.8: Normalized DRAM Energy (lower is better).

Fig. 5.8 shows the normalized dynamic energy in DRAM. Even in workloads with limited speedup, *METAL-IX* reduces DRAM Energy (e.g., $\approx 1.5 \times$ in Nest.SEL and WHERE, compared to address cache). *METAL-IX* reduces the total number of access by short-circuiting walks. In memory-bound workloads (Scan, SpMM in Fig. 5.8:④), we see a maximum reduction of DRAM energy. X-Cache’s hit path maximizes short-circuiting to the leaves, but the walk starts from the root on a miss. Thus, X-Cache has minimal traffic benefit over the address cache. *METAL-IX* caches intermediate nodes, maximizes reach, and saves traffic. In JOIN, *METAL-IX* experiences high contention as it targets multiple B+Trees. Here, *METAL-IX* short-circuits less and hence achieves less traffic reduction.

5.2.3 Energy

Result: *METAL* reduces cache energy by short-circuiting walks and reducing the number of accesses- $3 \times$ lower energy.

Fig. 5.9: Top compares the energy of cache organizations. Energy = per-access cost \times #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 $2-4 \times$. Compared to X-Cache, *METAL* achieves a higher hit rate by caching high 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. 5.9: Bottom, breaks down the energy of different modules: compute tile, *IX-cache*, walker logic + pattern controller [47]. We show representative workloads from each of the DSAs. The *IX-cache* accounts for $\frac{1}{3}$ of overall on-chip energy.

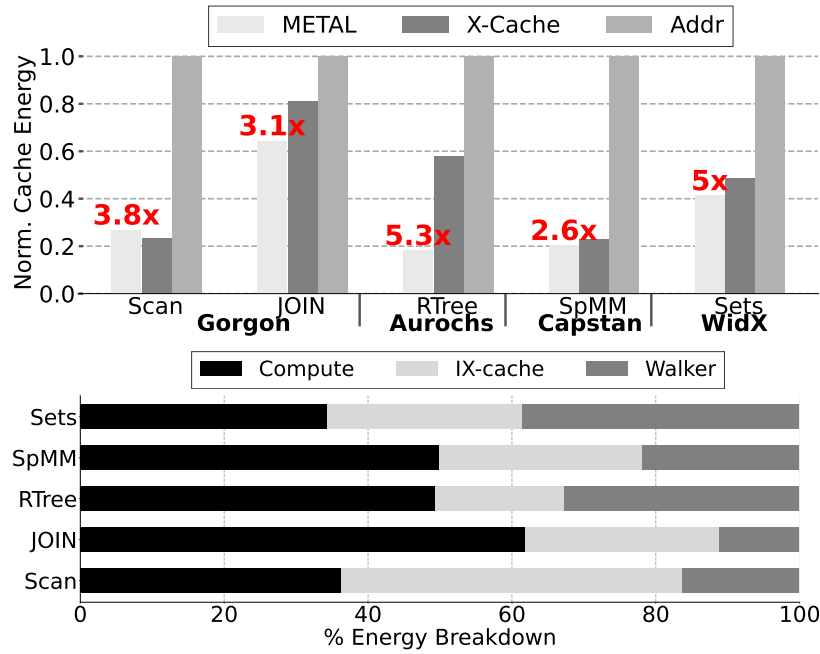


Figure 5.9: **Top:** Cache Energy. (Red: Cache access reduction relative to address-cache.) **Bottom:** Energy Breakdown.

5.2.4 Set-Associative vs. Fully-Associative IX-Cache

In the coming experiments, we refer to Sets as KVStore.

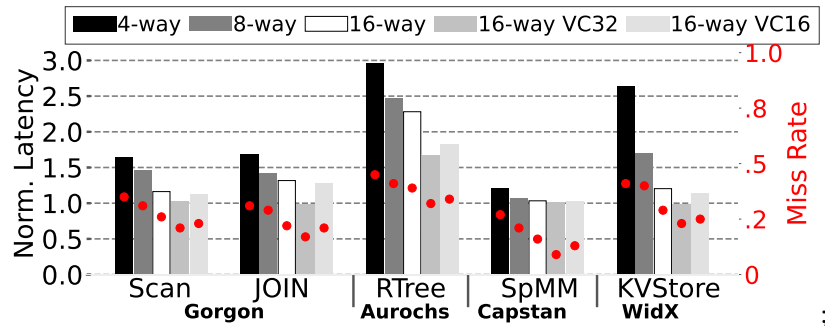


Figure 5.10: *IX-cache* Associativity. Left Y: Exe. Time Right Y: Miss rate. Lower is Better. Baseline: Fully Assoc.

Result: A 16-way *IX-cache* is only 15% slower compared to fully-associative; a 32 entry victim cache closes the gap.

We next characterize the impact of associativity. Fig. 5.10 plots the walk latency (lower is better) from 4-way to 16-way normalized to fully associative. i) In workloads with high reuse (SpMM), associativity has minimal impact. A 4-way increases walk latency 20% over FA, and an 8-way is comparable to FA. ii) Overall, the 8-way and 16-way caches increase walk latency 30% and 15%, respectively, over FA. iii) A 16-entry victim cache enables the 16-way to close the gap to within 5%. A 32-entry victim achieves the same performance.

RTree is the exception that overloads the *IX-cache*; performance is still 50% slower than fully-associative.

Key Block Size We now study the impact of the key ranges in the block (Fig. 5.11). We vary the block range from 64 to 1024 aligned keys and investigate the impact on performance. We find that 256 wide blocks are the most effective. In our workloads, we find that most index ranges are $\simeq 256$ wide. Thus, the 256 wide block encounters low set conflicts and avoids block duplication. We find that 1024 wide blocks lead to higher set conflicts as more index nodes map to the same set. This leads to a high miss rate, $\simeq 36\%$ vs. 12 % for the 256 wide blocks. 64 wide blocks are too narrow to be duplicated across multiple sets. The replication leads to cache space wastage and low utilization since the block only services a narrower range; nearly 45% of the cache capacity is lost.

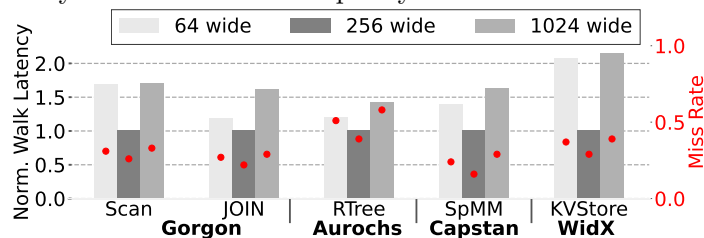


Figure 5.11: Impact of key block size. Baseline = 256 wide.

Cache entry size vs. index node size We now study the impact of smaller *IX-cache* entry that is $\frac{1}{2}$ the size of an index node (32 Bytes as opposed to 64 Bytes). Consequently, we can only cache a sector of the index node. We make the following observations: i) In some applications, even if *IX-cache* can only hold a sub-sector of the node, the effectiveness does not reduce, e.g., for scan, latency only goes up 25%. This is because sub-sectors may service multiple walks and maintain the hit rate. ii) In workloads with high spatial locality (multiple keys touched) at the leaf, caching only sub-sectors in the leaf impacts performance i.e., KVStore (sets) and SpMM prefer wider cache entries.

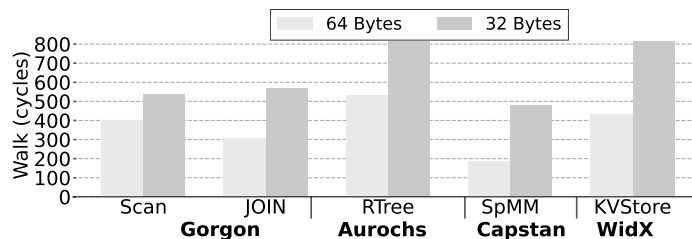


Figure 5.12: Impact of reducing *IX-cache* entry size.

5.2.5 Shared vs. Private *IX-cache*

Result: Walk latency increases by 6%—95% for private. Between 9%–62% of cache capacity lost due to duplication.

We investigate the impact of varying the number of tiles that share the *IX-cache*. Reducing the number of tiles shared reduces the potential for cooperative caching across walks. Instead, we have to replicate entries across each private cache, leading to a loss of effective capacity. The baseline is 256kB *IX-cache* shared by 64 tiles. We compare it against 3 configurations: 2 x 128kB (32 tiles share 128kB), 4 x 64kB and 8x 32kB. All configurations run *METAL-IX*. Fig. 5.13 plots the slowdown in execution time (or increase in walk latency), normalized to the baseline. We find that by privatizing the cache, we lose effective capacity (red numbers on the bars). In applications with immediate reuse (SpMM), the loss of effective capacity is minimal. In applications with index reuse and cooperative walks (RTree), privatizing leads to increased replication and loss of effective capacity. Here, the walk latency increases by 3 - 5 × even with *METAL-IX*.

We find that the loss of effective capacity leads to a 6% to 94% increase in walk latency. The loss of effective capacity ranges from 38% to 91%. The R-tree workload is the most sensitive to the loss of effective capacity. The R-tree workload has a high number of unique index entries and, thus, is most sensitive to the loss of effective capacity. The SpMM workload is the least sensitive to the loss of effective capacity. The SpMM workload has a high number of affine accesses and, thus, is least sensitive to the loss of effective capacity. Private *IX-cache* lacks cooperative caching, due to which we observe a performance degradation of 27.8% for 2 x 128k cache, 42.6% for 4 x 64k caches, and 61.4% for 8 x 32k caches (from figure 5.13). We observe that for SpMM, utilization is not affected much due to a high number of affine accesses, with a finite life span for each entry. In figure 5.13, the bars represent the increase in walk latency, and the numbers in red represent the effective capacity (or number of unique entries) in the *IX-cache*.

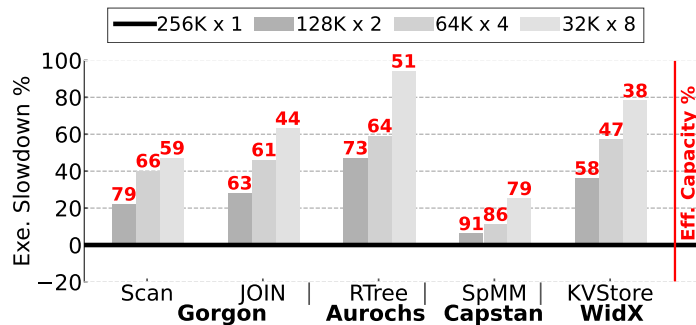


Figure 5.13: Private. vs Shared *IX-cache*. Lower is better.

Chapter 6

Summary & Thoughts

Indexed data structures have always been crucial, but their importance has grown significantly due to key technological advancements and data management trends, motivating their deployment and usage on DSAs. We have developed *METAL-IX*, an architectural addition to enable DSAs to manage and reuse indexes. Previous work used to cater to indexes in different formats, even resorting to asymptotically sub-optimal algorithms that are easier to accelerate[59]. *METAL-IX* contributes the *IX-cache*, a novel cache architecture that uses key indexes as cache tags for short-circuiting, thereby reducing latency for index reads. This allows us to port traditional indexes to DSAs efficiently. Previously, we used different DSAs to perform data analytics, spatial analytics, PageRank, etc. Now, with *METAL*, we can support the above workloads through a single dataflow architecture, thereby generalizing the dataflow architecture for different types of indexes.

The proliferation of sparse data demanded increased workloads that more commonly performed writes to the indexes. Many real-world applications such as graph processing [36, 56, 66, 3], machine learning [49, 42], and data analytics [17] involve dataset changes over time and often frequently. However, current state-of-the-art DSAs do not include a synchronization facility to support writes and just batch updates on the host machine. Therefore, we introduced *RBlox*, which developed a novel hardware synchronization facility for DSAs to support mutating *dynamic data-structures*. Unlike traditional address-based locks, *RBlox* utilizes ranges to concisely and symbolically represent synchronization boundaries. We also show that range locks are more efficient in hardware. To summarise, *RBlox* contributes in 2 major ways: i) *RBlox* reduces the number of locks in the critical path, and ii) *RBlox* relaxes ordering constraints in the chain of fine-grain locks.

Bibliography

- [1] Dennis Abts, John Kim, Garrin Kimmell, Matthew Boyd, Kris Kang, Sahil Parmar, Andrew Ling, Andrew Bitar, Ibrahim Ahmed, and Jonathan Ross. The groq software-defined scale-out tensor streaming multiprocessor: From chips-to-systems architectural overview. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–69, 2022.
- [2] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. Leap scratchpads: Automatic memory and cache management for reconfigurable logic. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 25–28, New York, NY, USA, February 2011. Association for Computing Machinery.
- [3] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Commongraph: Graph analytics on evolving data. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 133–145, 2023.
- [4] Tutu Ajayi, Vidya A Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B Kahng, Minsoo Kim, Jeongsup Lee, Uday Mallappa, Marina Neseem, et al. Toward an open-source digital flow: First learnings from the openroad project. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–4, 2019.
- [5] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37:1–24, 2003.
- [6] Saman Ashkiani, Martin Farach-Colton, and John D Owens. A dynamic hash table for the gpu. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 419–429. IEEE, 2018.
- [7] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. Practice of streaming processing of dynamic graphs: Concepts, models, and systems. *IEEE Transactions on Parallel and Distributed Systems*, 34(6):1860–1876, 2021.
- [8] Maciej Besta, Florian Mareending, Edgar Solomonik, and Torsten Hoefler. Slimsell: A vectorizable graph representation for breadth-first search. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 32–41. IEEE, 2017.
- [9] N. V. Vijaya Krishna Boppana and Saiyu Ren. A low-power and area-efficient 64-bit digital comparator. *J. Circuits Syst. Comput.*, 25(12), 2016.
- [10] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. Scalable and robust latches for database systems. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, pages 1–8, 2020.

- [11] Stephen Chou and Saman Amarasinghe. Compilation of dynamic sparse tensor algebra. *Int'l Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2022.
- [12] Chang Chua and R.B.N. Kumar. An improved design and simulation of low-power and area efficient parallel binary comparator. *Microelectron. J.*, page 84–88, aug 2017.
- [13] Eric S Chung, James C Hoe, and Ken Mai. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *PROC of the 19th FPGA*. ACM Request Permissions, February 2011.
- [14] Jason Clemons, Chih-Chi Cheng, Iuri Frosio, Daniel R Johnson, and Stephen W Keckler. A patch memory system for image processing and computer vision. In *Proc. of the 49th MICRO*, pages 1–13, 2016.
- [15] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. Towards general purpose acceleration by exploiting common data-dependence forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 924–939, 2019.
- [16] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. Towards general purpose acceleration by exploiting common data-dependence forms. In *Proc. of the 52nd MICRO*, pages 924–939, 2019.
- [17] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The {rocksdb} experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49, 2021.
- [18] Fabio Frustaci, Stefania Perri, Marco Lanuzza, and Pasquale Corsonello. Energy-efficient single-clock-cycle binary comparator. *Int. J. Circuit Theory Appl.*, 40(3):237–246, 2012.
- [19] Christina Giannoula, Nandita Vijaykumar, Nikela Papadopoulou, Vasileios Karakostas, Ivan Fernandez, Juan Gómez-Luna, Lois Orosa, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. Synchron: Efficient synchronization support for near-data-processing architectures. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 263–276. IEEE, 2021.
- [20] Goetz Graefe. Hierarchical locking in b-tree indexes. In *On Transactional Concurrency Control*, pages 45–73. Springer, 2007.
- [21] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th annual IEEE/ACM international symposium on microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.
- [22] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–333, 2019.

- [23] John L. Hennessy and David A. Patterson. Computer architecture: A quantitative approach, 2024. Section 7.2: Page 543.
- [24] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The art of multi-processor programming*. Newnes, 2020.
- [25] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.
- [26] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 600–614, 2019.
- [27] Hideaki Kimura, Goetz Graefe, and Harumi A Kuno. Efficient locking techniques for databases on modern hardware. In *ADMS@ VLDB*, pages 1–12. Citeseer, 2012.
- [28] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.
- [29] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin T Lim, and Parthasarathy Ranganathan. Meet the walkers: accelerating index traversals for in-memory databases. In *Proc. of the 46th MICRO*, pages 468–479, 2013.
- [30] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszal, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. *Spatial: a language and compiler for application accelerators*. ACM, New York, New York, USA, June 2018.
- [31] Alex Kogan, Dave Dice, and Shady Issa. Scalable range locks for scalable address spaces and beyond. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.
- [32] Tobin J Lehman and Michael J Carey. A study of index structures for main memory database management systems. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1985.
- [33] Ching-Kai Liang and Milos Prvulovic. Misar: Minimalistic synchronization accelerator with resource overflow management. *ACM SIGARCH Computer Architecture News*, 43(3S):414–426, 2015.
- [34] Weifeng Liu and Brian Vinter. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 339–350, 2015.
- [35] Devavret Makkar, David A Bader, and Oded Green. Exact and parallel triangle counting in dynamic graphs. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 2–12. IEEE, 2017.

- [36] Tim Mattson, David Bader, Jon Berry, Aydin Buluc, Jack Dongarra, Christos Faloutsos, John Feo, John Gilbert, Joseph Gonzalez, Bruce Hendrickson, Jeremy Kepner, Charles Leiserson, Andrew Lumsdaine, David Padua, Stephen Poole, Steve Reinhardt, Mike Stonebraker, Steve Wallach, and Andrew Yoo. Standards for graph algorithm primitives. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–2, 2013.
- [37] Robert McColl, Oded Green, and David A Bader. A new parallel algorithm for connected components in dynamic graphs. In *20th Annual International Conference on High Performance Computing*, pages 246–255. IEEE, 2013.
- [38] Mike O’Connor. Highlights of the high- bandwidth memory. <https://docplayer.net/206066-Highlights-of-the-high-bandwidth-memory-hbm-standard.html>.
- [39] Tayo Oguntebi and Kunle Olukotun. Graphops: A dataflow library for graph analytics acceleration. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 111–117, 2016.
- [40] Oracle. Selecting an index strategy.
- [41] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. Terrace: A hierarchical graph container for skewed dynamic graphs. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1372–1385, 2021.
- [42] Jongsoo Park, Sheng Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. Faster cnns with direct sparse convolutions and guided pruning. *arXiv preprint arXiv:1608.01409*, 2016.
- [43] Stefania Perri and Pasquale Corsonello. Fast low-cost implementation of single-clock-cycle binary comparator. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 55(12):1239–1243, 2008.
- [44] Kerttu Pollari-Malmi, Eljas Soisalon-Soininen, and Tatu Ylonen. Concurrency control in b-trees with batch updates. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):975–984, 1996.
- [45] Raghu Prabhakar, Sumti Jairath, and Jinuk Luke Shin. Sambanova sn10 rdu: A 7nm dataflow architecture to accelerate software 2.0. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 350–352. IEEE, 2022.
- [46] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matthew Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A Reconfigurable Architecture For Parallel Paterns. In *Proc. of the 44th ISCA*, pages 389–402, 2017.
- [47] Aditya Prasanna. Spot: Rethinking sparse data locality with smart cursors and reuse patterns, Theses (M.Sc.)–Simon Fraser University, 2024.
- [48] Martin Quinson and Flavien Vernier. Byte-range asynchronous locking in distributed settings. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 191–195. IEEE, 2009.

- [49] Samyam Rajbhandari, Yuxiong He, Olatunji Ruwase, Michael Carbin, and Trishul Chilimbi. Optimizing cnns on multicores for scalability, performance and goodput. *ACM SIGARCH Computer Architecture News*, 45(1):267–280, 2017.
- [50] Redis, 2023. <https://redis.io/docs/about/>.
- [51] Samuel Rogers, Joshua Slycord, Mohammadreza Baharani, and Hamed Tabkhi. Gem5-SALAM: A system architecture for LLVM-based accelerator modeling. In *Proc. of the 53rd MICRO*, pages 471–482, 2020.
- [52] Alexander Rucker, Matthew Vilim, Tian Zhao 0001, Yaqi Zhang 0001, Raghu Prabhakar, and Kunle Olukotun. Capstan: A vector RDA for sparsity. In *Proc. of the 54th MICRO*, pages 1022–1035, 2021.
- [53] Karthikeyan Sankaralingam, Tony Nowatzki, Vinay Gangadhar, Preyas Shah, Michael Davies, William Galliher, Ziliang Guo, Jitu Khare, Deepak Vijay, Poly Palamuttam, Maghawan Punde, Alex Tan, Vijay Thiruvengadam, Rongyi Wang, and Shunmiao Xu. The mozart reuse exposed dataflow processor for ai and beyond: industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022.
- [54] Ali Sedaghati, Milad Hakimi, Reza Hojabr, and Arrvindh Shriraman. X-cache: a modular architecture for domain-specific caches. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 396–409, 2022.
- [55] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. Palm: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors. *Proceedings of the VLDB Endowment*, 4(11):795–806, 2011.
- [56] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.
- [57] Piyush Tyagi and Rishikesh Pandey. High-speed and area-efficient scalable N-bit digital comparator. *IET Circuits, Devices & Systems*, 14(4):450–458, 2020.
- [58] Enrique Vallejo, Ramon Beivide, Adrian Cristal, Tim Harris, Fernando Vallejo, Osman Unsal, and Mateo Valero. Architectural support for fair reader-writer locking. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 275–286. IEEE, 2010.
- [59] Matthew Vilim, Alexander Rucker, Yaqi Zhang 0001, Sophia Liu, and Kunle Olukotun. Gorgon: Accelerating machine learning from relational data. In *Proc. of the 47th ISCA*, 2020.
- [60] Matthew Vilim, Alexander Rucker, and Kunle Olukotun. Aurochs: An architecture for dataflow threads. In *Proc. of the 48th ISCA*, 2021.
- [61] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 461–472. IEEE, 2018.

- [62] Jian Weng, Jian, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. DSAGEN: Synthesizing programmable spatial accelerators. In *Proc. of the 47th ISCA*, pages 268–281, 2020.
- [63] Ayse Yilmazer and David Kaeli. Hql: A scalable synchronization mechanism for gpus. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 475–486. IEEE, 2013.
- [64] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. Gamma: Leveraging gustavson’s algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2021*, page 687–701, New York, NY, USA, 2021. Association for Computing Machinery.
- [65] Lixin Zhang, Zhen Fang, and John B Carter. Highly efficient synchronization based on active memory operations. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 58. IEEE, 2004.
- [66] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 293–302. IEEE, 2017.
- [67] Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, and Guang R Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 35–45, 2007.