ManyRVData Project Report

CachePool: Manycore Cluster of Customizable, Lightweight Scalar-Vector PEs for Irregular Data-Plane Workloads

Report by:
Diyou Shen, Ph.D. Student
Zexin Fu, Ph.D. Student
dishen, zexifu@iis.ee.ethz.ch

Supervised by:
Prof. Luca Benini
Prof. Alessandro Vanelli-Coralli
lbenini, avanelli@iis.ee.ethz.ch

Integrated Systems Laboratory,
ETH Zürich,
Zürich, Switzerland

August 29, 2025

Contents

1 Executive Summary

This document reports on the *ManyRVData* project. The aim of the project is to design and evaluate a scalable, cache-based manycore architecture with scalar-vector capability for the 5G Radio Access Networks (RAN) dataplane. The focus is on efficiently supporting the RLC layer, which requires both high-throughput packet processing and control parameter computation under tight latency constraints. The architecture is designed to maintain high performance and energy efficiency when scaling to hundreds of Processing Elements (PEs), each operating on large, sparse, and irregular data structures typical of dataplane workloads.

The project explores a heterogeneous manycore cluster in which lightweight, in-order scalar RISC-V cores handle control-intensive tasks, while vector engines excel at data-intensive workloads, all connected through a shared, cache-based memory hierarchy. To better match the unpredictable memory access patterns in RLC packet processing, the design abandons scratchpad-only approaches (such as in MemPool) and avoids hardware cache coherence entirely. Instead, it employs a **shared L1 cache** with **configurable private L1 partitions** per PE or per group of PEs, enabling flexibility in bandwidth allocation and minimizing contention while keeping implementation complexity low.

The architecture is designed to address three representative RLC dataplane scenarios. The first is a **single-user peak-rate case**, representing the extreme throughput demand of a single access user and serving as a stress test for per-user performance, latency, and sustained packet processing under continuous load. The second is a **multi-user peak-load case**, where a moderate number of active users generate high aggregate throughput, testing the system's ability to scale across many threads while maintaining low contention and predictable latency. The third is a **multi-user average-load case**, reflecting a large-scale deployment with thousands of concurrent users at more typical per-user rates, emphasizing fairness, resource isolation, and stable long-term operation. In the first year, the project will concentrate on the single-user peak-rate case to validate the core architecture, memory subsystem, and scalar-vector processing efficiency before extending evaluation to multi-user scenarios in the second stage.

The project spans two years, with the first-year activities organized into three main Work Packages (WPs):

- **WP1 – Scalable Cache Design for a Many-core Cluster:** Define the architecture specification and explore design options, followed by the initial design of key building blocks. Carry out preliminary implementation and power–performance–area (PPA) analysis.
- **WP2 – Processing Element: Microarchitecture and ISA Optimization:** Specify the microarchitecture and prototype a non-blocking load/store interface, together with instruction set extensions (vector unit) for dataplane processing workloads and control parameter computation.
- **WP3 – HW-SW Mapping, Runtime Management, and Benchmarking:** Conduct preliminary benchmarking with representative dataplane packet-processing kernels, and develop prototype programming APIs and runtime support for deployment, allocation, and scheduling.

Building on ETH Zürich's experience with the *MemPool* architecture, the *ManyRVData* design transitions from software-managed scratchpads to a hardware-managed, shared cache system, while keeping the per-core architecture simple and scalable. This approach provides high flexibility in resource partitioning and efficient support for sparse, irregular workloads without the area and energy costs of full hardware coherence.
The first year will deliver the following milestones:

- **M1.1 – Preliminary ManyRVData Cluster Design (M12):** Release of open-source hardware and documentation.

- **M1.2 – Preliminary Benchmark Suite and Analysis (M12):** Release of open-source software and documentation, accompanied by a benchmarking report.

The design source code for hardware and software will be made available as open source, as described in Section **??**.

2 Introduction

Wireless baseband dataplane processing is a prime domain for energy-efficient parallel computing: the RAN market is vast, and the throughput and efficiency requirements are escalating. As 5G evolves toward 5.5G and 6G, per-cell computational complexity is projected to grow far beyond what technology scaling alone can deliver, demanding architectural and implementation advances across the stack. This project addresses that challenge with a focus on dataplane tasks in the RLC layer. Service Data Units (SDUs) from higher layers are queued and assembled into Protocol Data Units (PDUs) with sequence headers for retransmission management before being passed to MAC, an interaction that tightly couples performance and reliability constraints.

Viewed as a computing workload, RLC exhibits three defining features. First, scale: modern deployments must sustain on the order of 10 million packets/s overall, with each RLC instance exceeding 10^3 packets/s and thousands of instances active concurrently. Second, latency: the Transition Time Interval (TTI) spans 62.5 µs to 1 ms (typically 500 µs), setting a hard real-time envelope for all sub-tasks. Third, irregularity: per-user tasks are independent but highly variable, the code footprint is small, while data structures (e.g., linked lists) create sparse, non-contiguous accesses over large address ranges. Together, these properties imply hundreds of concurrent threads touching dispersed state with poor spatial locality.

TODO: Add SoA comparison here

For such access patterns, explicitly managed hierarchies (e.g., scratchpad memory + DMA) are ill-suited: tiling and double-buffering become inefficient as data dependencies branch unpredictably. A hardware-managed, cache-based hierarchy is required; however, enabling hundreds of PEs to access shared cache banks efficiently is challenging. Using hardware-managed private caches would require costly hardware cache-coherence maintenance. To address this, we propose a shared L1 cache with a configurable, software-managed private-L1 partition design.

TODO: whole project overview **ManyRVData** explores a scalable, cache-based manycore architecture built around the following key design elements.

1. *Shared L1 with configurable, software-managed private partitions.*

   By default, all L1 cache partitions are shared and accessible by all PEs, connected through *hierarchical interconnects* to maximize bandwidth and minimize latency.

   For data that is heavily accessed by a specific tile and sensitive to latency or bandwidth contention, software can reconfigure selected L1 partitions to be private to that tile; in this mode, cache coherence for these private partitions is maintained entirely by software. This private partition reduces average access latency, increases effective bandwidth for the owner tile, and improves predictability of timing under the tight TTI constraints of the workload.

2. *Heterogeneous scalar–vector execution with fine-grain interleaving.*

   The cluster combines lightweight, in-order scalar RISC-V cores for control-intensive tasks with tightly coupled vector engines for data-intensive kernels. Scalar and vector execution are closely integrated, with negligible switching overhead between scalar and vector instructions, making the architecture particularly suitable for workloads that interleave scalar–vector operations at fine granularity.

3. *Scalable L1 cache controller for high in-flight miss handling.*

   An L1 cache controller that scales to handle many in-flight misses by embedding MSHR metadata directly into the cache-line SRAM arrays, eliminating the need for dedicated MSHR registers and allowing the theoretical number of outstanding misses to match the number of cache lines.

This enables high in-flight cache miss handling, which is beneficial for the target workloads, e.g., sparse linked-list traversal and per-user buffer/reassembly lookups, where many independent misses need to be in flight to hide memory latency and sustain throughput.

Here we summarize the main contributions achieved in the first year of the *ManyRVData* project. Guided by the work package plan, our efforts have spanned microarchitecture design, RTL implementation, memory-system innovation, physical-aware evaluation, and software–hardware co-design for representative dataplane workloads. Our contributions in the first year include: **TODO:** whole project overview

1. **Scalar–vector PE microarchitecture.** We designed the *Snitch* lightweight in-order scalar core tightly coupled to the *Spatz* vector engine. This integration allows scalar control and vector data phases to interleave at fine granularity with near-zero handoff overhead, enabling efficient execution of workloads with frequent scalar–vector switching.

2. **Tile-level RTL integration.** We implemented a parameterizable tile consisting of multiple Snitch+Spatz core complexes, connected to multiple shared L1 cache banks via a low-latency interconnect. The interconnect supports configurable memory address interleaving to match workload patterns. DRAMSys integration is included to enable cycle-accurate simulation of real-world DRAM behavior.

3. **Novel L1 cache controller.** We developed an L1 cache controller that embeds MSHR metadata directly into the cache-line SRAM, removing the need for a fixed-size MSHR file. This design allows the number of outstanding misses to scale with the number of cache lines, supporting deep hit-under-miss and secondary-miss merging, well suited for sparse, pointer-chasing workloads such as linked-list traversal and per-user buffer/reassembly lookups.

4. **Physical-aware PPA evaluation.** We performed power, performance, and area analysis using physical-aware synthesis in a 12 nm technology node, providing realistic insights into implementation cost and scalability.

5. **Dataplane kernel implementation and evaluation.** We implemented and benchmarked a RLC data management kernel, which includes key features such as linked-list traversal, vectorized data movement, spin-lock synchronization, and memory management runtime, along with vector compute kernels such as `dotp`, `gemv`, and `gemm`. These kernels serve both as performance drivers for architecture evaluation and as a foundation for future runtime and application development.

3 CachePool Architecture

The previous works [1, 2] focus on scaling the number of PEs for higher performance, together with Scratchpad Memory (SPM)-based shared L1 and Direct Memory Access (DMA) engine for data movement. However, such a scaling pattern struggles to handle the high bandwidth requirement from RLC workload and lacks flexibility in programming. Thus, a more suitable architecture is to adopt a cache-based memory hierarchy with a bandwidth-aware scaling pattern.

CachePool is a highly flexible heterogeneous many-core architecture designed with a fully shared multi-banked L1 cache, connecting to the multi-channel DRAM/HBM off-chip main memory. It ensures high bandwidth and performance while keeping cache coherence effective and lightweight.

3.1 Processing Elements

3.1.1 Snitch Scalar Core

CachePool is equipped with single-stage, 32 bit RISC-V *Snitch* cores [3] as its scalar and host cores, with the support for RISCV32I base Instruction Set Architecture (ISA). The instructions are decoded and executed inside the core, while the vector or floating-point related instructions are offloaded to the vector core. To prevent the read-after-write (RAW) hazard, a counter tracks the execution of vector memory-related instructions, preventing Snitch from loading before the vector storing is completed.

Similar to MemPool and TeraPool, the Snitch's Load Store Unit (LSU) in CachePool contains a scoreboard that keeps track of outstanding memory accesses and allows Snitch to proceed with instructions as long as there are no RAW hazards. The LSU supports multiple outstanding memory requests, allowing Snitch to issue a series of loads and stores without waiting for the response [1] to tolerate latency in memory hierarchies. Given the Non-Uniform Memory Access (NUMA) interconnection design and main memory behavior, Snitch's scoreboard retires loads out-of-order while guaranteeing in-order delivery to the execution units. The number of supported outstanding transactions is parameterizable and can be tuned depending on the maximum memory access latency within the Cluster's L1 memory.

Snitch features an accelerator port to offload vector and floating-point instructions to the Spatz vector core. The architecture view of Snitch core with the Spatz vector core is shown in Figure 1. In the following section, we will introduce the architecture of the Spatz vector core, with the optional floating-point support.

3.1.2 Spatz Vector Core

A 32-bit streamlined vector core Spatz [4] is tightly integrated with the Snitch core inside the core-complex to enhance the performance and improve energy efficiency.

The architecture of Spatz is based on the RVV version 1.0 with a generic accelerator interface. It has a 2 KiB latch-based Vector Register File (VRF) to provide high-energy-efficient vector processing. Spatz supports both vector and scalar floating-point calculations with an additional Floating Point Unit (FPU) Sequencer. The scalar floating-point instructions reuse the FPUs inside the Vector Functional Unit (VFU) to minimize the extra FPU area costs. The floating-point feature can be turned off for specific application requirements to save area and energy.

Figure 1 shows a block diagram of the vector core. A scoreboard inside the controller provides the dependency and vector chaining support of the Spatz vector core. It has its specific Vector Load Store Unit (VLSU) with built-in reorder buffers, providing the ability of multiple outstanding memory accesses in a NUMA architecture. The small latch-based VRF reduces the area and energy consumption, providing the possibility of integrating it as a building block of a manycore architecture.

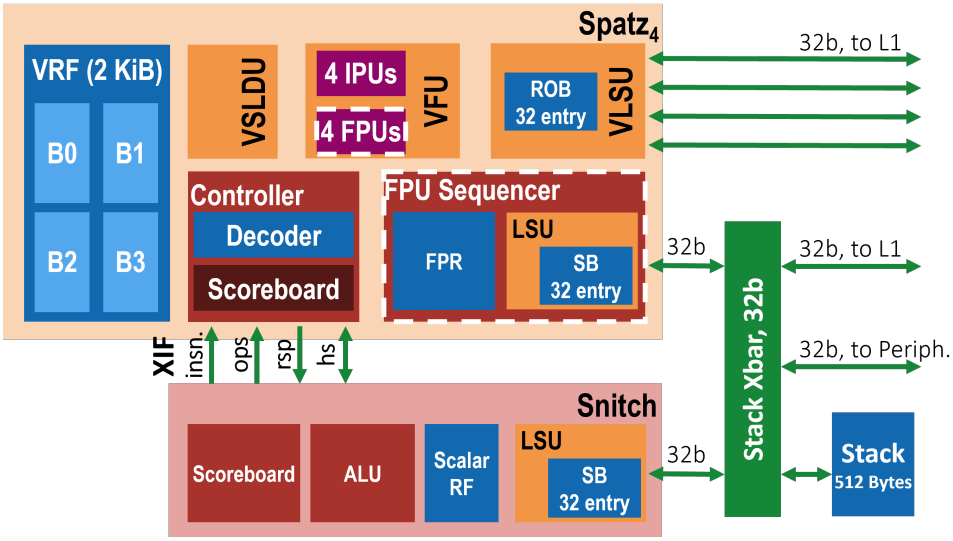

Figure 1: CachePool Core-Complex (CC): a Snitch core tightly coupled with a Spatz vector core. The dashed-lined modules can be disabled in the configuration.

3.2 Tile and Interconnection

3.2.1 Computing Tile

The Cluster is constructed hierarchically to ensure the feasibility of CachePool's physical design. The basic building block of CachePool is the *Tile* shown in Figure 2, which is designed for massive replication. It contains 4 32-b Snitch-Spatz CCs, each equipped with a small 512 Byte private Stack memory.

The CCs share a 8 KiB four-way set-associative L1 Instruction Cache (I$). The Advanced eXtensible Interface (AXI) master port, used for L1 I$ refill, is shared between all the cores.

For the key data memory side, CCs are tightly coupled to one shared 256 KiB four-banked L1 data cache through a set of runtime-configurable crossbars, which will be detailed in section 3.2.2. Each cache bank is equipped with an In-Situ cache controller 3.4 addressing a range of the main memory space.

3.2.2 Interconnection Module

3.2.3 Atomic Unit

3.3 Physically Feasible Hierarchical Interconnection Design

3.4 Memory Subsystem

3.4.1 Cache Hierarchy and Organization

3.4.2 Cache Controller

3.5 System Level Design

3.5.1 DRAMsys Co-Simulation

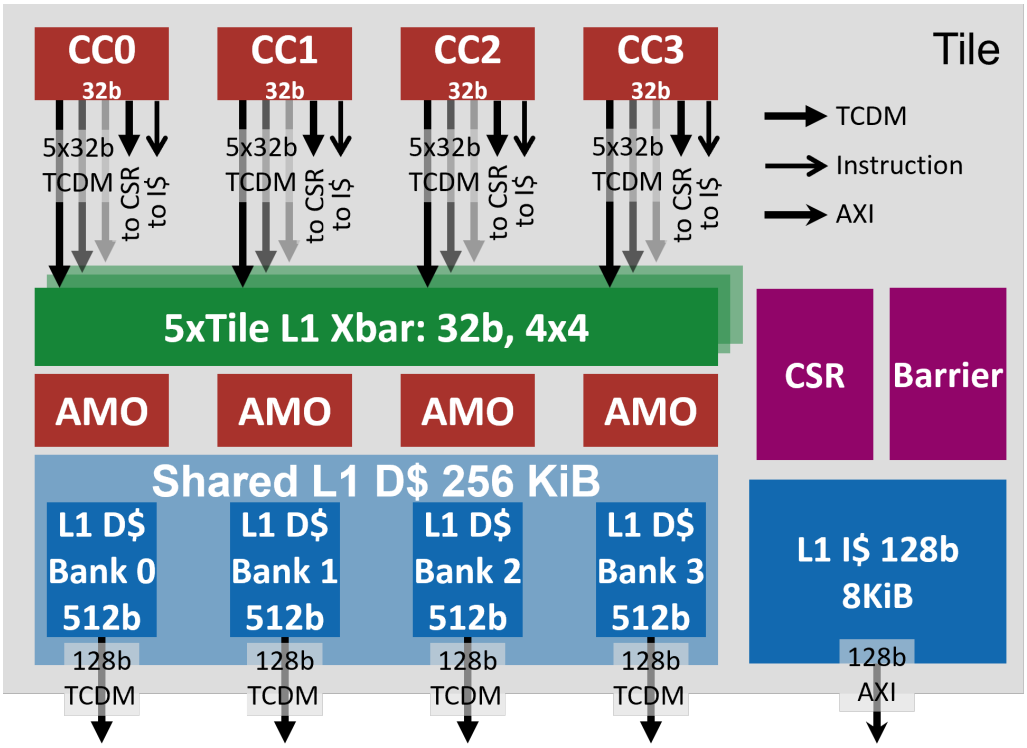

Figure 2: Overview of the CachePool Tile architecture with 256KiB L1 Cache. The remote interconnection between tiles is not shown.

4 Physical Implementation

4.1 Methodology

4.2 Floorplan

4.3 Power, Performance and Area

5 CachePool Programming Model

5.1 Compiler Support and Runtimes

5.2 Fast Hardware Barrier Synchronization

6 Vector Kernels

6.1 DOTP

6.2 GEMM

6.3 GEMV

6.4 Exponent

7 RLC Kernel Implementation

This section presents the software implementation of the RLC kernels used in the ManyRVData dataplane. We focus on downlink AM-mode processing at gNodeB side under high-load traffic models, aligned with the Layer-2 DP scenarios and timing target provided by the partner specifications. In our DP setting, RLC entities operate under tight TTI constraints (typ. 0.5 ms), must sustain multiple million packets per second throughput, and manipulate irregular data structures (linked lists), yielding sparse and non-contiguous memory accesses. These properties favor a cache-based, hardware-managed memory hierarchy rather than software-managed DMA and scratchpad memory. *ManyRVData* therefore targets a shared-cache cluster. By default, L1 cache banks are shared across all cores, with an optional per-tile private L1 partition for stronger locality or isolation.

7.1 Workload model and constraints

We adopt three traffic scenarios for evaluation (single-user peak, multi-user peak, multi-user average). They determine per-slot packet counts per RLC entity, and consequently the concurrency and memory pressure seen by the kernels. The partner document defines packet sizes, per-second slot counts, and closed-form relationships between throughput and packet rate, which we reuse in our test harness.

Table 1: RLC evaluation scenarios (DL/UL). Values shown are from the partner's L2DP description and serve as our default harness configuration.

Scenario	Users	DL Gbps	DL pkt B	DL pps	DL slots/s	DL pkts/slot	UL Gbps	UL pkts/slot
Single-user peak	1	20	1350	1,851,851	1600	1157	1	1953
Multi-user peak	48	50	800	7,812,500	1600	4882	8	15625
Multi-user average	4800	20	800	3,125,000	1600	1963	4	7813

Notes. Per-user rates derive from the total divided by the number of users; slot counts per second are fixed by numerology; TB sizes for assembly come from the control/scheduler stage and are provided to the RLC assembly task at runtime.

Protocol and configuration. All entities run in Acknowledged Mode (AM) with 18-bit sequence numbers (SN); `pollPDU`=32 and `pollByte`=25,000. In our testcases, gNB receives only *positive* ACK STATUS PDUs; retransmission and timers are disabled.

7.2 Software Architecture and Implementation

7.2.1 Workflow of the RLC Kernel

The RLC kernel can be abstracted as a producer–consumer program: producers generate and enqueue work items; the consumer processes them and maintains protocol state. The end-to-end workflow and how the two sides interact are shown in Fig. **??**.

Producers repeatedly pull the next PDCP SDU descriptor from a shared cursor (under a small lock), request a `Node` from the fixed-size pool (`mm_alloc()` under allocator lock), fill its metadata (`data`, `tgt`, `len`), and append it to the tail of the shared *to-send* list (`list_push_back()` under list lock). If allocation fails or no new SDU is available, the producer enters the *Wait* state (short backoff) and retries.

A single *consumer* thread loops on the *to-send* list: it removes the head node (`list_pop_front()` under lock); if the list is empty, it briefly *Wait*s. Otherwise, it performs the data move (vectorized memcpy standing in for AMD PDU assembly) from `data` to `tgt`, then updates RLC TX bookkeeping: increments `pduWithoutPoll` and `byteWithoutPoll`, advances the next sequence number `vtNext`, and accounts the SDU leaving/entering queues by decrementing `sduNum`/`sduBytes` on

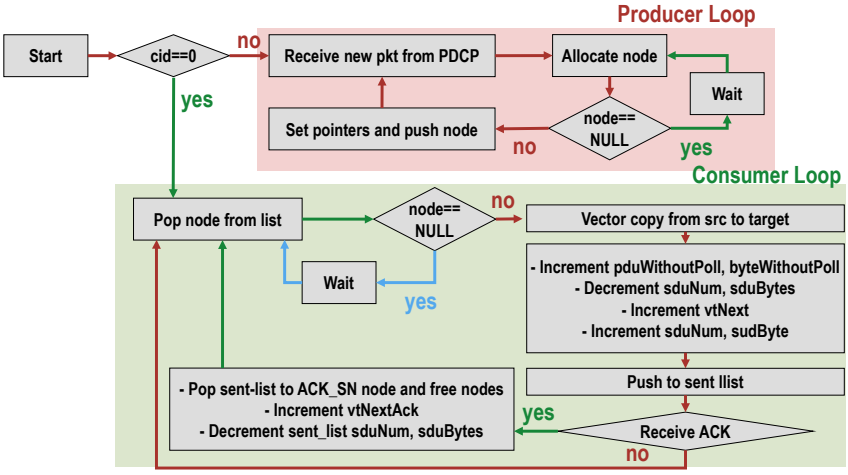

Figure 3: High-level producer–consumer architecture for RLC kernel

the *to-send* list and incrementing the same counters on the *sent* list. The processed node is appended to the *sent* list under lock. Upon reception (or simulation) of a STATUS with `ACK_SN`, the consumer pops acknowledged nodes from the head of the *sent* list up to `ACK_SN`, advances `vtNextAck`, decrements the *sent*-list counters accordingly, and returns each acknowledged node to the pool via `mm_free()`.

All shared structures (PDCP cursor, allocator, both linked lists) are protected by compact spinlocks to ensure atomic push/pop and allocation; a one-time cluster barrier precedes the workflow so producers and consumer start from a consistent state. Back-pressure naturally arises at two points shown as *Wait*: empty queues (consumer idle) and pool exhaustion (producers back off). The loop continues until producers exhaust PDCP SDUs and the consumer drains and acknowledges all in-flight nodes.

7.2.2 Producer–Consumer Roles

Producers. Producer threads simulate packet ingress (e.g., PDCP SDUs) and enqueue work items into a shared *to-send* queue implemented as a doubly linked list. Each producer repeatedly: (i) obtains the next PDCP packet descriptor from a shared cursor, (ii) allocates a `Node` from a fixed-size memory pool, (iii) fills its metadata (payload pointer, size), and (iv) appends it to the tail of the *to-send* list under a lock. This models concurrent ingress sources feeding the RLC layer.

Consumer. A single consumer thread represents the RLC transmitter. It repeatedly: (i) pops a `Node` from the head of the *to-send* list under a lock, (ii) performs data movement (copy/assemble) to the target buffer (standing in for AMD PDU assembly), (iii) updates RLC state (e.g., sequence counters; bytes/PDUs without poll), and (iv) pushes the `Node` into a *sent* list (tracking in-flight PDUs) under a lock. The consumer also simulates ACK processing: once an in-flight threshold is reached, it dequeues a batch from the *sent* list (front), advances the transmit window, and returns those `Node`s to the memory pool.

7.2.3 Concurrency and Synchronization

Spinlocks. Shared resources (*to-send* list, *sent* list, memory pool, and the PDCP package cursor) are protected by simple spinlocks implemented with atomic test-and-set (*busy-wait*). List operations use tight spinning (low-latency critical sections), while the memory allocator adds a small backoff delay to reduce contention. A hardware barrier synchronizes cores once after initialization to ensure a consistent start.

7.2.4 Data Structures and Memory Management

Double linked lists. Work queues are `LinkedList` (head/tail, counters) of `Node`s. `list_push_back()` appends in $O(1)$ time (updates tail), `list_pop_front()` removes in $O(1)$ time (updates head).

Node. Each `Node` holds `prev/next` pointers, a pointer to the source payload, a pointer to the destination buffer, and the payload length. Per-node locks are initialized but list-level locks suffice for this workload.

Fixed-size pool allocator. A global memory context manages a statically allocated, cacheline-aligned buffer partitioned into fixed-size *pages* (one page per `Node`). Allocation uses a lock-protected bump pointer; when the pool is exhausted, freed pages are recycled from a lock-protected free-list. This yields deterministic, fragmentation-free allocation without a general-purpose heap.

7.2.5 Task Scheduling and Thread Management

For now, we use a static, bare-metal mapping: each core executes *one* function (producer or consumer). Coordination is via shared queues and spinlocks. Back-pressure arises naturally from the finite memory pool and queue occupancy.

8 Benchmarking and Performance Analysis

8.1 Methodology

8.2 Vector Kernel Benchmarking

8.3 RLC Kernel Benchmarking

9 Next Year Plan

10 Conclusion

Acknowledgments

References

[1] Samuel Riedel et al. "MemPool: A Scalable Manycore Architecture With a Low-Latency Shared L1 Memory". In: *IEEE Transactions on Computers* 72.12 (2023), pp. 3561–3575. DOI: 10.1109/TC.2023.3307796.

[2] Yichao Zhang et al. "TeraPool-SDR: An 1.89TOPS 1024 RV-Cores 4MiB Shared-L1 Cluster for Next-Generation Open-Source Software-Defined Radios". In: *Proceedings of the Great Lakes Symposium on VLSI 2024*. GLSVLSI '24. Clearwater, FL, USA: Association for Computing Machinery, 2024, pp. 86–91. ISBN: 9798400706059. DOI: 10.1145/3649476.3658735. URL: https://doi.org/10.1145/3649476.3658735.

[3] Florian Zaruba et al. "Snitch: A Tiny Pseudo Dual-Issue Processor for Area and Energy Efficient Execution of Floating-Point Intensive Workloads". In: *IEEE Transactions on Computers* 70.11 (Nov. 2021), pp. 1845–1860. ISSN: 1557-9956. DOI: 10.1109/TC.2020.3027900.

[4] Matteo Perotti et al. "Spatz: Clustering Compact RISC-V-Based Vector Units to Maximize Computing Efficiency". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 44.7 (2025), pp. 2488–2502. DOI: 10.1109/TCAD.2025.3528349.