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

September 9, 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

Previous works [1, 2] primarily scale the number of PEs to increase performance, relying on an Scratchpad Memory (SPM)-based shared L1 memory together with a Direct Memory Access (DMA) engine for data movement. While effective at small to medium scales, this approach struggles to sustain the bandwidth demands of RLC workloads and offers limited programming flexibility. To address these challenges, we explore a cache-based memory hierarchy combined with a bandwidth-aware scaling strategy.

CachePool is a heterogeneous many-core architecture built around a fully shared, multi-banked L1 cache connected to multi-channel DRAM/HBM main memory. This design delivers high bandwidth and performance, while maintaining cache coherence in an efficient and lightweight manner.

3.1 Processing Elements

3.1.1 Snitch Scalar Core

CachePool employs single-stage, 32 bit RISC-V *Snitch* cores [3] as scalar and host processors, supporting the RV32I base Instruction Set Architecture (ISA). The cores execute scalar instructions directly, while vector and floating-point instructions are offloaded to the attached Spatz vector core. To prevent read-after-write (RAW) hazards, a dedicated counter tracks in-flight vector memory operations, ensuring that Snitch stalls loads until pending vector stores have completed.

As in MemPool and TeraPool, the Snitch Load Store Unit (LSU) integrates a scoreboard to monitor outstanding memory operations, enabling forward progress as long as no RAW hazards occur. The LSU supports multiple outstanding requests, allowing Snitch to issue a sequence of loads and stores without waiting for responses [1], thereby tolerating long memory latencies. Owing to the Non-Uniform Memory Access (NUMA)-style interconnection and main memory behavior, the scoreboard retires loads out-of-order while guaranteeing in-order delivery to the execution pipeline. The maximum number of outstanding transactions is parameterizable and can be tuned according to the expected memory latency of the cluster's shared L1 cache.

Each Snitch core features an accelerator port to offload vector and floating-point instructions to the Spatz vector core. Figure 1 illustrates the core complex architecture, showing a Snitch core tightly coupled with its Spatz counterpart. Dashed-lined modules indicate optional components that can be disabled at configuration time. The next section introduces the Spatz vector core in more detail, including its optional floating-point support.

3.1.2 Spatz Vector Core

The 32-bit streamlined vector core *Spatz* [4] is tightly coupled with the Snitch scalar core within the core complex, boosting both performance and energy efficiency.

Spatz implements the RVV 1.0 specification through a generic accelerator interface and features a 2 KiB latch-based Vector Register File (VRF) optimized for energy-efficient vector processing. It supports both vector and scalar floating-point operations, coordinated by an additional Floating Point Unit (FPU) sequencer. To minimize area overhead, scalar floating-point instructions reuse the FPUs embedded in the Vector Functional Unit (VFU). For area- or energy-constrained applications, floating-point support can be selectively disabled.

The architectural view of Spatz is shown in Figure 1. A scoreboard within the controller provides dependency tracking and enables vector chaining. Spatz includes a dedicated Vector Load Store Unit (VLSU) equipped with reorder buffers, allowing multiple outstanding memory requests in the NUMA environment. The compact latch-based VRF further reduces area and energy, making Spatz a scalable building block for many-core architectures well-suited.

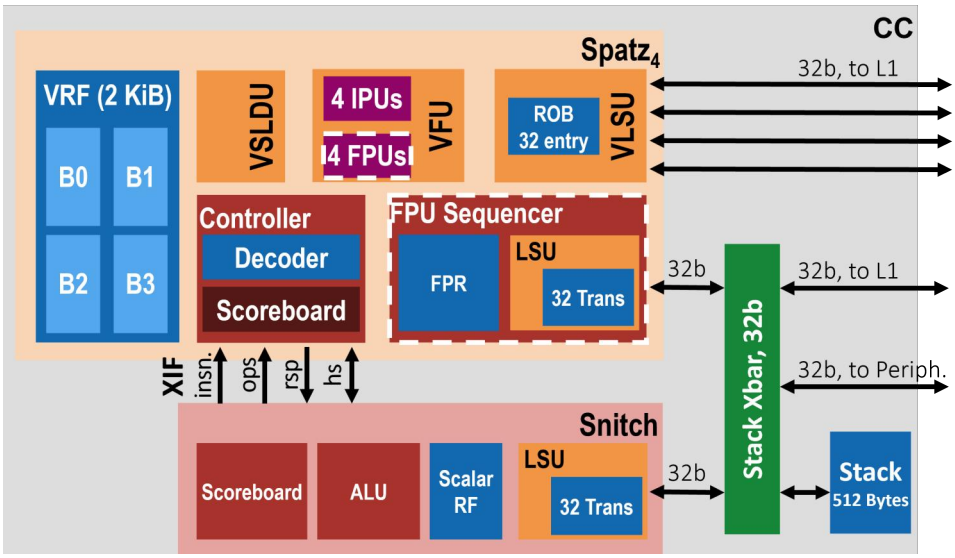


Figure 1: CachePool Core-Complex (CC): a Snitch core tightly coupled with a Spatz vector core. Dashed-lined modules represent optional components that can be disabled in the configuration.

3.2 Tile

3.2.1 Computing Tile

CachePool adopts a hierarchical cluster organization to ensure scalability and physical design feasibility. The fundamental building block is the *Tile*, shown in Figure 2, which is designed for massive replication. Each Tile integrates 4 32-bit Snitch–Spatz CCs, each equipped with a private 512 B stack memory. The number of cores per Tile is configurable through the system configuration files.

All CCs within a Tile share an 8 KiB four-way set-associative L1 Instruction Cache (I$). A single Advanced eXtensible Interface (AXI) master port handles the refill traffic for this instruction cache, shared among all cores.

On the data side, the CCs are tightly coupled to a shared 256 KiB four-banked L1 Data Cache (D$) via runtime-configurable crossbars, described in detail in Section **??**. Each cache bank is managed by an in-situ cache controller (Section 3.4), which maps to a portion of the global memory address space. Cache refills are served to the cluster through a Tightly Coupled Data Memory (TCDM)-based protocol.

3.2.2 Atomic Unit

Although the single-level shared-cache design eliminates the need for complex cache coherence, memory consistency must still be ensured. To this end, an atomic unit is placed before the L1 D$ controller, providing Atomic Memory Operations (AMOs) in compliance with the RISC-V "A" standard extension.

This unit was designed initially for SPM-based systems, where memory access latency is highly predictable. For CachePool, it was adapted with a handshake-driven Finite State Machine (FSM) to cope with the variable latencies introduced by cache-based memory access.

3.3 Physically Feasible Hierarchical Interconnection Design

3.3.1 L1 Cache Interconnection

A hierarchical topology is essential to implement a physically feasible interconnection network between PEs and the fully shared L1 D$. To sustain low-latency TCDM access, the design

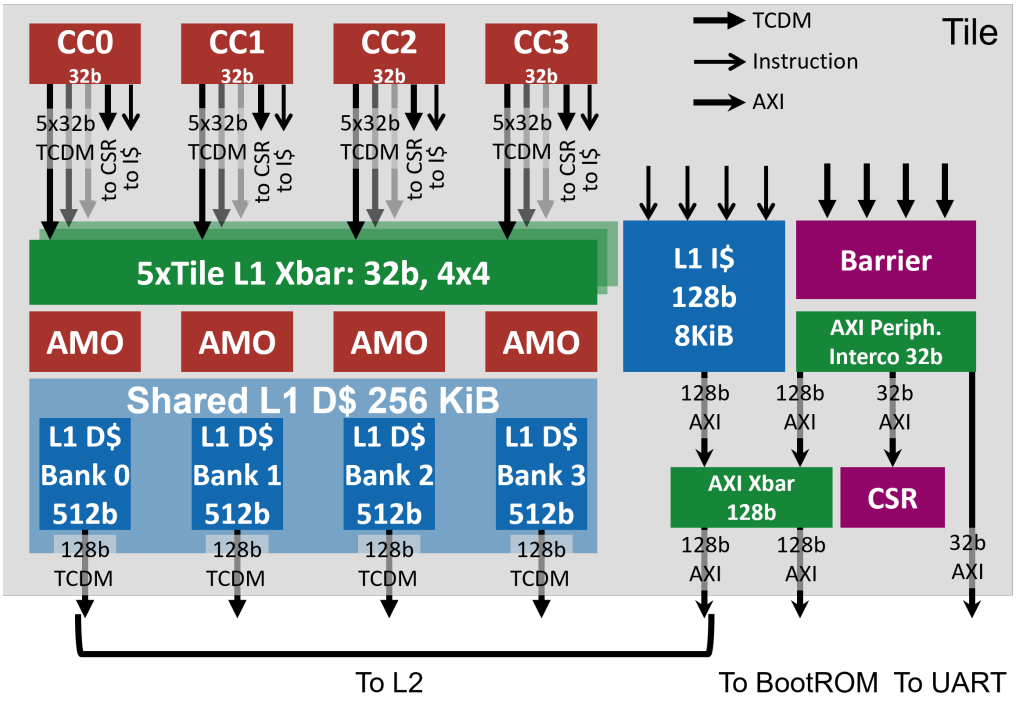


Figure 2: Overview of the CachePool Tile architecture with 256KiB L1 Cache. The remote interconnection between tiles is not shown.

employs fully combinational routing with logarithmic crossbars and arbiters at each hierarchy level. The interconnection node follows the principles described in [5].

For physical implementation, the L1 crossbar is decomposed into five independent 4×4 32-bit crossbars, reducing routing pressure and improving layout feasibility. Bank interleaving for the L1 D$ is integrated within the crossbar. Rather than embedding complex address remapping logic inside the cache controller, a lightweight address scrambling mechanism is applied at the L1 crossbar, configured via Control Status Registers (CSRs). This approach is inspired by the DAS design [6], which introduced software-controlled address remapping in the TeraPool architecture. Our implementation is simplified, as the cache controller in CachePool can directly access the entire memory address space.

3.3.2 L2 Refill Interconnection

A refill crossbar connects the cache refill interface to the off-chip DRAM channels at the cluster level. To bridge the mismatch between the cacheline width (512 bit) and the interconnection width (128 bit), burst request support is added to the TCDM interconnect. Unlike the previous TCDM-Burst work [7], which targeted bandwidth-limited systems with constrained physical resources, CachePool provides sufficient memory bandwidth to sustain core consumption. Here, the wider cacheline acts similarly to a "next-line prefetcher," primarily serving to hide DRAM access latency. As a result, burst responses in CachePool are forwarded serially rather than grouped.

The refill crossbar implements a select-lock mechanism to enforce ordering within a burst. This mechanism guarantees in-order delivery of burst responses, significantly reducing the cache controller's logic complexity.

In the current single-tile implementation, the refill crossbar only routes traffic from the cache controller to the DRAM channels through an all-to-all topology (Figure 3a). In future multi-tile designs, traffic from multiple tiles must be multiplexed, making a hierarchical interconnection essential. Figure 3b illustrates a possible four-tile organization, where a two-level structure,

similar to the L1 crossbar, minimizes routing overhead. Looking ahead, the refill crossbar could also be integrated into each Tile to simplify cluster-level routing further.

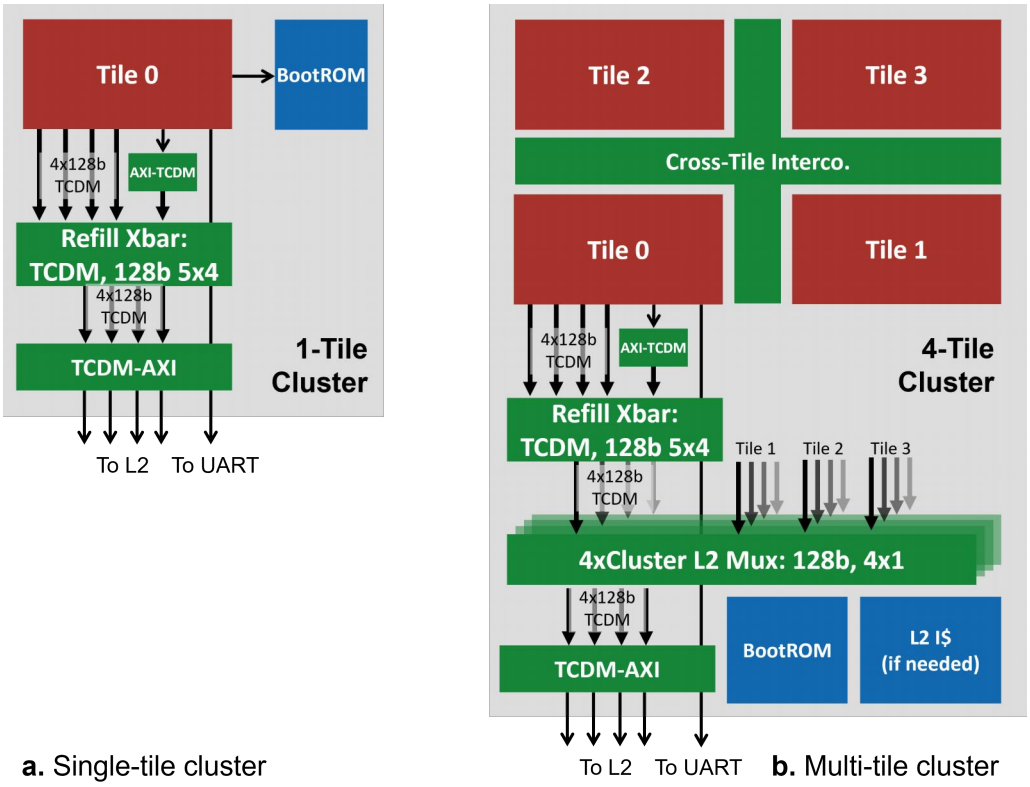


Figure 3: Overview of the CachePool cluster design. (a) Single-tile implementation with direct crossbar refill. (b) A possible four-tile extension with a hierarchical crossbar is needed for physical feasibility.

3.3.3 Cross-Tile Interconnection

A cross-tile interconnection becomes necessary as the cluster scales beyond a single Tile to enable data access across neighboring Tiles. This can be realized with an all-to-all crossbar for a small number of Tiles. However, as the system scales further, a Nework on Chip (NoC) interconnection becomes more suitable to maintain performance and physical feasibility. In the current first-phase implementation, CachePool targets a single-Tile cluster, and therefore, no cross-tile interconnection has been included yet.

3.4 Memory Subsystem

3.4.1 Cache Hierarchy and Organization

CachePool adopts a fully shared L1 cache design without an intermediate L2 level. This choice eliminates the need for coherence management and avoids data duplication across cache levels. However, it increases interconnection complexity, since direct communication paths between Tiles are required (Section 3.3).

Alternatively, we are exploring an architecture with small private write-through caches at each PE, backed by a shared L2. This organization simplifies coherence, since a lightweight snoopy-style invalidation network would suffice. A comparative evaluation of these two approaches will be carried out once the write-through L1 design is completed.

3.4.2 InSitu Cache Controller

The shared L1 cache is managed by the *InSitu Cache* controller [8], which tolerates long-latency accesses directly to DRAM. InSitu Cache is a non-blocking, high-performance design that avoids the need for additional SRAM structures such as write buffers or Miss Handling Architectures (MHAs).

Its efficiency comes from reusing otherwise wasted cache resources for miss handling:

- Unified in-cacheline miss handling for both read and write misses.
- A Dynamic Subentry Expansion (DSE) mechanism that allocates multiple cache lines to a single miss entry, extending miss handling capacity under high-latency conditions.
- A Dynamic Subset Least Recent Used (DS-LRU) replacement policy that reclaims wasted cache lines for miss handling, while keeping the critical path short enough for high-frequency implementation.

Further details on the InSitu Cache controller will be provided once the related work is published.

3.4.3 DRAMSys Co-Simulation

A time-efficient yet cycle-accurate Dynamic Random-Access Memory (DRAM) subsystem is essential for TeraPool cluster co-simulation to evaluate data transfer performance alongside high-level memory hierarchies. *DRAMSys5.0* [9], an open-source and flexible framework for DRAM design space exploration, meets these requirements.[1] Developed by the Microelectronic Systems Design Research Group at RPTU Kaiserslautern-Landau, *DRAMSys5.0* is based on SystemC TLM-2.0 and supports cycle-accurate modeling of multiple memory technologies, including DDR3/4, LPDDR4, Wide I/O 1/2, GDDR5/5X/6, and HBM1/2.

During the TeraPool project, an Register Transfer Level (RTL) interface to *DRAMSys5.0*[2] was developed and connected to the cluster via the AXI interconnect. The *DRAMSys5.0* subsystem was compiled into a shared library (`.so` format) and linked into the ModelSim environment,[3]. In CachePool, we reuse the same infrastructure with a default DDR4 configuration featuring four memory channels.

[1] The DRAMSys5.0 repository is available at `https://github.com/tukl-msd/DRAMSys`, released under the BSD 3-Clause License.

[2] `https://github.com/pulp-platform/dram_rtl_sim`

[3] ModelSim supports behavioral, RTL, and gate-level simulation with a platform-independent compile flow: `https://eda.sw.siemens.com/en-US/ic/modelsim/`.

4 Physical Implementation

This section evaluates the physical feasibility and area efficiency of different CachePool single-tile cluster configurations, with particular focus on the impact of varying cacheline widths.

4.1 Methodology

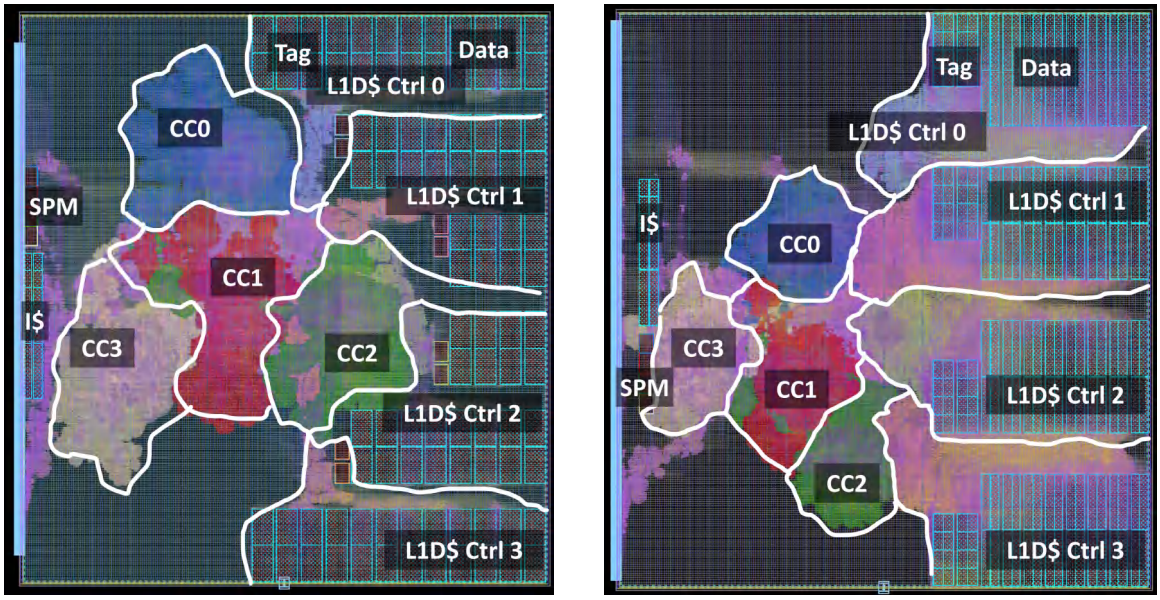
The CachePool cluster is implemented in GlobalFoundries' 12LPPLUS FinFET technology. Once the design flow is fully validated, we plan to migrate to TSMC's 7 nm process. For the single-tile cluster, a hierarchical implementation is not required; however, such a flow will become essential as the design scales to multi-tile systems. All synthesis, placement, and routing are carried out using Synopsys Fusion Compiler 2023.12-SP4 in a flattened flow. Future scaled-up clusters will adopt a top-down hierarchical approach. The current design targets 1 GHz at worst-worst PVT corner for the 12LPPLUS process.

4.2 Floorplan

Since the current technology node is not the final target, the floorplan is not tuned for optimal density. Instead, the backend results primarily serve to assess physical feasibility and provide a reference for logic area estimation.

Figures **??** and **??** illustrate the placed-and-routed layouts of the single-tile CachePool cluster for cacheline widths of 128 b and 512 b, respectively. Both configurations integrate a four-banked 256 KiB L1 D$.

Due to the unavailability of 128×512 b SRAM macros, the 512 b configuration is emulated using four 128×128 b SRAM instances. This workaround results in a larger area footprint than a dedicated macro would require. A more detailed breakdown is presented in the next section.



128 b cacheline configuration with FPUs. 512 b cacheline configuration without FPUs.

Figure 4: Placed-and-routed layouts of the CachePool single-tile cluster for two configurations: (left) 128 b cacheline with FPU, (right) 512 b cacheline without FPU.

4.3 Area Breakdown

The Gate Equivalent (GE), defined as the area of a two-input NAND gate, is a widely used metric to represent logic area in physical design independently of the technology node. The main area breakdown is derived from post-synthesis data, since the floorplan used in Figure 5 is not fully optimized. We also report a breakdown from post-layout results in Figure 6, but these numbers are more sensitive to floorplan variations and should be interpreted with caution.

The dominant contributor to the total area is the L1 memory subsystem, which accounts for roughly 60% of the total in the 128 b cacheline configuration with FPU support. Increasing the cacheline size leads to a significant growth in both the data banks and the controller. The data bank overhead arises from the lack of native 128×512 SRAM macros, which forces us to emulate them using four 128×128 SRAMs. This results in poor area efficiency. Two possible remedies are under consideration:

- *Multi-muxed SRAMs:* A 128×512 configuration can be realized by multiplexing narrower SRAM macros. While still suboptimal, this approach improves efficiency compared to the current solution.
- *Bandwidth-matched SRAMs:* The effective core-to-cache bandwidth is only 128 b. Wider cachelines are mainly used for pseudo prefetching and bandwidth matching with DRAM. Therefore, using a 512×128 SRAM macro could achieve full core utilization without performance loss, while improving area efficiency.

The PEs together occupy about 30% of the total area, with approximately 900 kGE attributable to the FPUs.

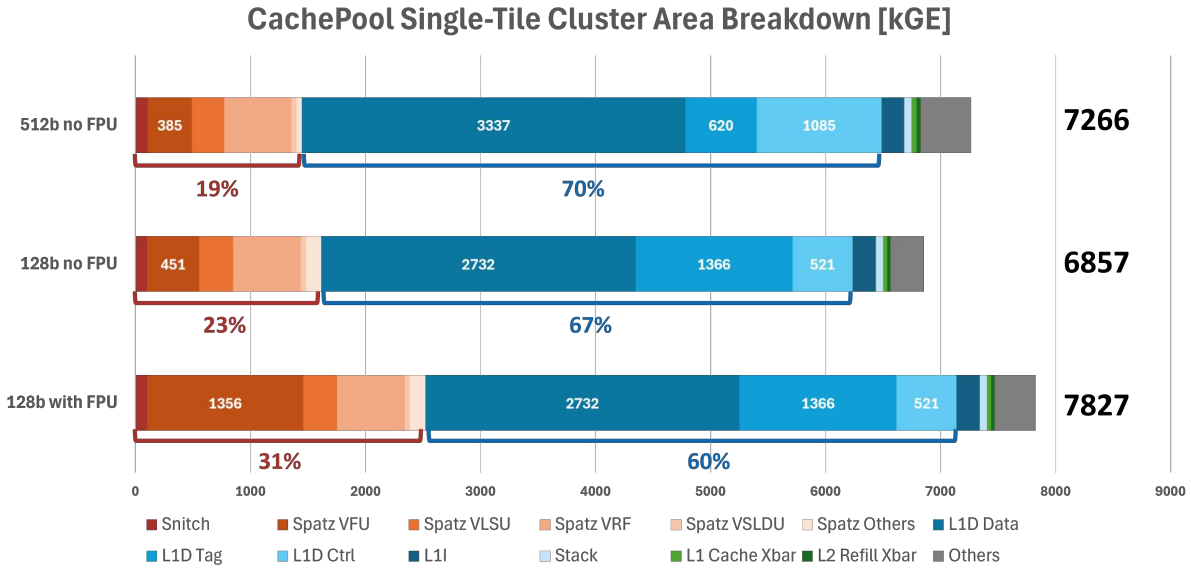


Figure 5: Post-synthesis area breakdown of the CachePool single-tile cluster. The 128 b configuration without FPU is estimated by subtracting the FPU area.

Overall, the total cluster area ranges from roughly 6.8 MGE (128 b cacheline without FPU) to 7.8 MGE (128 b cacheline with FPU), with the 512 b cacheline variant occupying about 7.3 MGE. This confirms that memory dominates the area budget, while FPUs contribute a significant but optional overhead. Interconnection logic and miscellaneous components remain below 5%, indicating they are not a limiting factor at this scale. Looking ahead, scaling CachePool to many tiles will be primarily constrained by SRAM efficiency rather than logic complexity, making optimized memory macros a key requirement for future implementations.

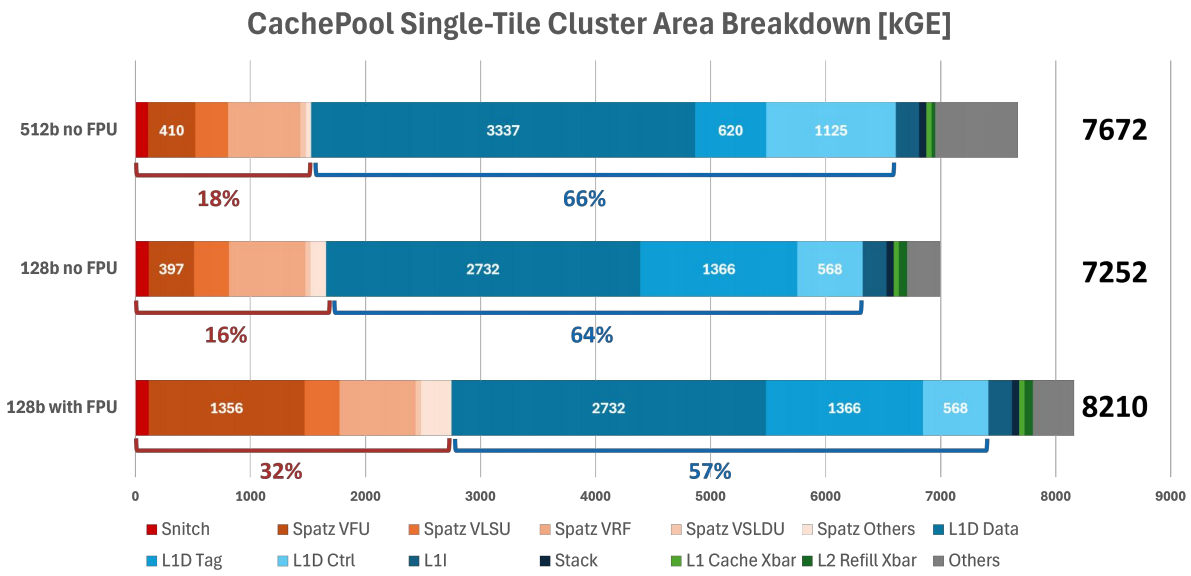


Figure 6: Post-route area breakdown of the CachePool single-tile cluster. The 128 b configuration without FPU is estimated by subtracting the FPU area.

5 Vector Kernels

5.1 DOTP

5.2 GEMM

5.3 GEMV

5.4 Exponent

6 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.

6.1 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.

6.2 Software Architecture and Implementation

6.2.1 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

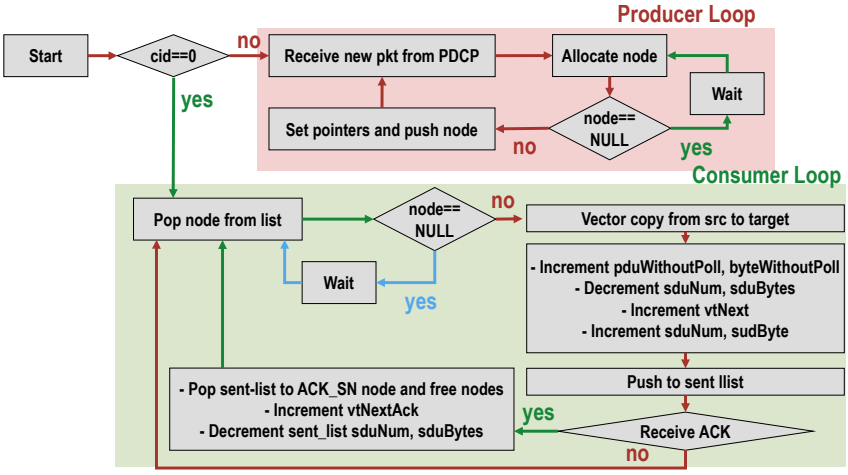


Figure 7: High-level producer–consumer architecture for RLC kernel

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.

6.2.2 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.

6.2.3 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.

6.2.4 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.

6.2.5 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.

7 Benchmarking and Performance Analysis

This section evaluates the performance of the CachePool single-tile cluster through cycle-accurate simulation using ModelSim, with *DRAMSys* serving as the main memory model. We investigate the impact of different cacheline widths (128 b, 256 b, and 512 b) under both hot- and cold-cache conditions. The simulations are carried out on the configurations with FPUs.

7.1 Vector Kernel Benchmarking

Figure 8 reports the execution cycles of representative vector kernels. Solid bars correspond to hot-cache runs, while the shaded extensions capture the additional cycles incurred under cold-cache conditions. In general, longer cachelines improve performance, particularly in the cold-cache case.

For the most memory-bound kernel (dot product), the 512 b configuration achieves over 20% FPU utilization even under cold-cache conditions, highlighting the cluster's ability to tolerate memory latency and exploit available off-chip bandwidth. As data reuse increases, cache misses have a smaller performance impact. For compute-bound kernels, such as matrix multiplication, the cluster reaches over 90% FPU utilization in both hot- and cold-cache scenarios, demonstrating that computation resources can be fully exploited when memory bandwidth is not the limiting factor.

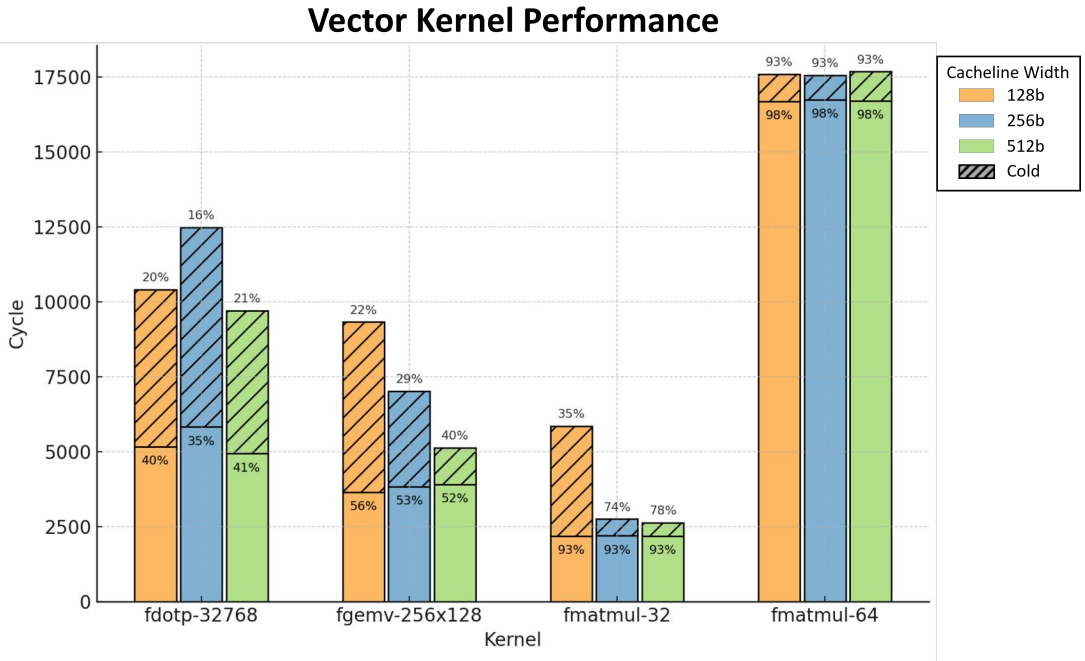


Figure 8: Vector kernel benchmark results of the CachePool cluster with different cacheline widths. Solid bars represent hot-cache execution, shaded extensions show additional cycles for cold-cache execution.

7.2 RLC Kernel Benchmarking

8 Next Year Plan

9 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.

[5] Abbas Rahimi et al. "A fully-synthesizable single-cycle interconnection network for Shared-L1 processor clusters". In: *2011 Design, Automation and Test in Europe*. 2011, pp. 1–6. DOI: 10.1109/DATE.2011.5763085.

[6] Bowen Wang et al. *A Dynamic Allocation Scheme for Adaptive Shared-Memory Mapping on Kilo-core RV Clusters for Attention-Based Model Deployment*. 2025. arXiv: 2508.01180 [cs.AR]. URL: https://arxiv.org/abs/2508.01180.

[7] Diyou Shen et al. "TCDM Burst Access: Breaking the Bandwidth Barrier in Shared-L1 RVV Clusters Beyond 1000 FPUs". In: *2025 Design, Automation and Test in Europe Conference (DATE)*. 2025, pp. 1–7. DOI: 10.23919/DATE64628.2025.10992996.

[8] Chi Zhang. *Pulp-Platform-Insitu-Cache*. URL: https://github.com/pulp-platform/Insitu-Cache.

[9] Lukas Steiner, Matthias Jung, Felipe S. Prado, et al. "DRAMSys4.0: An Open-Source Simulation Framework for In-depth DRAM Analyses". In: *International Journal of Parallel Programming* 50 (April 2022 2022), pp. 217–242. DOI: 10.1007/s10766-022-00727-4. URL: https://doi.org/10.1007/s10766-022-00727-4.

A Scalar instructions

TeraPool supports the *zfinx* and *zhinx* standard RISC-V extensions. Additionally, it implements also the following *smallfloat* extensions with the same rationale: the floating point unit reads and writes from the integer register file. A complete list of the smallfloat extensions and a detailed explanation of the acronyms used below can be found at `https://iis-git.ee.ethz.ch/smach/smallFloat-spec`.

31-25	24-20	19-15	14-12	11-7	6-0	
funct7	rs2	rs1	funct3	rd	opcode	R-type

31-27	26-25	24-20	19-15	14 13-12	11-7	6-0	
rs3	fc2	rs2	rs1	rm*	rd	opcode	R4-type

31-20	19-15	14 13-12	11-7	6-0	
imm[11:0]	rs1	funct3	rd	opcode	I-type

31-25	24-20	19-15	14 13-12	11-7	6-0	
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

31-30	29-25	24-20	19-15	14	13-12	11-7	6-0	
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode	RVF-type

RV32Xf16 Half-Precision Floating-Point Extension, bit[26,25]=10 (binary16)

rs3	10	rs2	rs1	rm*	rd	1000011	FMADD.H	$(rs1 * rs2) + rs3$
rs3	10	rs2	rs1	rm*	rd	1000111	FMSUB.H	$(rs1 * rs2) - rs3$
rs3	10	rs2	rs1	rm*	rd	1001011	FNMSUB.H	$-(rs1 * rs2) + rs3$
rs3	10	rs2	rs1	rm*	rd	1001111	FNMADD.H	$-(rs1 * rs2) - rs3$
0000010		rs2	rs1	rm*	rd	1010011	FADD.H	$rs1 + rs2$
0000110		rs2	rs1	rm*	rd	1010011	FSUB.H	$rs1 - rs2$
0001010		rs2	rs1	rm*	rd	1010011	FMUL.H	$rs1 * rs2$
0001110		rs2	rs1	rm*	rd	1010011	FDIV.H	$rs1/rs2$
0101110		00000	rs1	rm*	rd	1010011	FSQRT.H	$\sqrt{rs1}$
0010010		rs2	rs1	000	rd	1010011	FSGNJ.H	rs1, sign of rs2
0010010		rs2	rs1	001	rd	1010011	FSGNJN.H	rs1, inv. sign of rs2
0010010		rs2	rs1	010	rd	1010011	FSGNJX.H	rs1, sign rs1 $\oplus$ sign rs2
0010110		rs2	rs1	000	rd	1010011	FMIN.H	min
0010110		rs2	rs1	001	rd	1010011	FMAX.H	max
1010010		rs2	rs1	010	rd	1010011	FEQ.H	equal
1010010		rs2	rs1	001	rd	1010011	FLT.H	less than
1010010		rs2	rs1	000	rd	1010011	FLE.H	less than or equal
1100010		00000	rs1	rm*	rd	1010011	FCVT.W.H	to sgn. word (32bit)
1100010		00001	rs1	rm*	rd	1010011	FCVT.WU.H	to usgn. word (32bit)
1101010		00000	rs1	rm*	rd	1010011	FCVT.H.W	from sgn. word (32bit)
1101010		00001	rs1	rm*	rd	1010011	FCVT.H.WU	from usgn. word (32bit)
1110010		00000	rs1	001	rd	1010011	FCLASS.H	classify

Conversions with F Standard Extension

0100000	00010	rs1	000	rd	1010011	FCVT.S.H	binary16 $\rightarrow$ binary32
0100010	00000	rs1	rm*	rd	1010011	FCVT.H.S	binary32 $\rightarrow$ binary16

RV32Xf8 Quarter-Precision Floating-Point Extension, bit[26,25]=11 (binary8)

imm[11:0]			rs1	000	rd	0000111	FLB	load
imm[11:5]		rs2	rs1	000	imm[4:0]	0100111	FSB	store
rs3	11	rs2	rs1	rm*	rd	1000011	FMADD.B	$(rs1 * rs2) + rs3$
rs3	11	rs2	rs1	rm*	rd	1000111	FMSUB.B	$(rs1 * rs2) - rs3$
rs3	11	rs2	rs1	rm*	rd	1001011	FNMSUB.B	$-(rs1 * rs2) + rs3$
rs3	11	rs2	rs1	rm*	rd	1001111	FNMADD.B	$-(rs1 * rs2) - rs3$
0000011		rs2	rs1	rm*	rd	1010011	FADD.B	$rs1 + rs2$
0000111		rs2	rs1	rm*	rd	1010011	FSUB.B	$rs1 - rs2$
0001011		rs2	rs1	rm*	rd	1010011	FMUL.B	$rs1 * rs2$
0001111		rs2	rs1	rm*	rd	1010011	FDIV.B	$rs1/rs2$
0101111		00000	rs1	rm*	rd	1010011	FSQRT.B	$\sqrt{rs1}$
0010011		rs2	rs1	000	rd	1010011	FSGNJ.B	rs1, sign of rs2
0010011		rs2	rs1	001	rd	1010011	FSGNJN.B	rs1, inv. sign of rs2
0010011		rs2	rs1	010	rd	1010011	FSGNJX.B	rs1, sign rs1 $\oplus$ sign rs2
0010111		rs2	rs1	000	rd	1010011	FMIN.B	min
0010111		rs2	rs1	001	rd	1010011	FMAX.B	max
1010011		rs2	rs1	010	rd	1010011	FEQ.B	equal
1010011		rs2	rs1	001	rd	1010011	FLT.B	less than
1010011		rs2	rs1	000	rd	1010011	FLE.B	less than or equal
1100011		00000	rs1	rm*	rd	1010011	FCVT.W.B	to sgn. word (32bit)
1100011		00001	rs1	rm*	rd	1010011	FCVT.WU.B	to usgn. word (32bit)
1101011		00000	rs1	rm*	rd	1010011	FCVT.B.W	from sgn. word (32bit)
1101011		00001	rs1	rm*	rd	1010011	FCVT.B.WU	from usgn. word (32bit)
1110011		00000	rs1	001	rd	1010011	FCLASS.B	classify

Conversions with F Standard Extension

0100000	00011	rs1	000	rd	1010011	FCVT.S.B	binary8 $\rightarrow$ binary32
0100011	00000	rs1	rm*	rd	1010011	FCVT.B.S	binary32 $\rightarrow$ binary8

Conversions with Xf16 Extension

0100010	00011	rs1	000	rd	1010011	FCVT.H.B	binary8 $\rightarrow$ binary16
0100011	00010	rs1	rm*	rd	1010011	FCVT.B.H	binary16 $\rightarrow$ binary8

Xfvec Vectorial Floating-Point Ext. with Xf16, FLEN$\geq$32, vfmt=10 (binary16)

10	00001	rs2	rs1	0	10	rd	0110011	VFADD.H	$rs1 + rs2$
10	00001	rs2	rs1	1	10	rd	0110011	VFADD.R.H	$rs1 + rs2$, R
10	00010	rs2	rs1	0	10	rd	0110011	VFSUB.H	$rs1 - rs2$
10	00010	rs2	rs1	1	10	rd	0110011	VFSUB.R.H	$rs1 - rs2$, R
10	00011	rs2	rs1	0	10	rd	0110011	VFMUL.H	$rs1 * rs2$
10	00011	rs2	rs1	1	10	rd	0110011	VFMUL.R.H	$rs1 * rs2$, R
10	00100	rs2	rs1	0	10	rd	0110011	VFDIV.H	$rs1/rs2$
10	00100	rs2	rs1	1	10	rd	0110011	VFDIV.R.H	$rs1/rs2$, R
10	00101	rs2	rs1	0	10	rd	0110011	VFMIN.H	min
10	00101	rs2	rs1	1	10	rd	0110011	VFMIN.R.H	min, R
10	00110	rs2	rs1	0	10	rd	0110011	VFMAX.H	max
10	00110	rs2	rs1	1	10	rd	0110011	VFMAX.R.H	max, R
10	00111	00000	rs1	0	10	rd	0110011	VFSQRT.H	$\sqrt{rs1}$
10	01000	rs2	rs1	0	10	rd	0110011	VFMAC.H	$(rs1 * rs2) + rd$
10	01000	rs2	rs1	1	10	rd	0110011	VFMAC.R.H	$(rs1 * rs2) + rd$, R
10	01001	rs2	rs1	0	10	rd	0110011	VFMRE.H	$(rs1 * rs2) - rd$
10	01001	rs2	rs1	1	10	rd	0110011	VFMRE.R.H	$(rs1 * rs2) - rd$, R
10	01100	00001	rs1	0	10	rd	0110011	VFCLASS.H	classify
10	01101	rs2	rs1	0	10	rd	0110011	VFSGNJ.H	rs1, sign of rs2
10	01101	rs2	rs1	1	10	rd	0110011	VFSGNJ.R.H	rs1, sign of rs2, R
10	01110	rs2	rs1	0	10	rd	0110011	VFSGNJN.H	rs1, inv. sign of rs2
10	01110	rs2	rs1	1	10	rd	0110011	VFSGNJN.R.H	rs1, inv. sign of rs2, R
10	01111	rs2	rs1	0	10	rd	0110011	VFSGNJX.H	rs1, sign rs1 $\oplus$ sign rs2
10	01111	rs2	rs1	1	10	rd	0110011	VFSGNJX.R.H	rs1, sign rs1 $\oplus$ sign rs2, R
10	10000	rs2	rs1	0	10	rd	0110011	VFEQ.H	equal
10	10000	rs2	rs1	1	10	rd	0110011	VFEQ.R.H	equal, R
10	10001	rs2	rs1	0	10	rd	0110011	VFNE.H	not equal
10	10001	rs2	rs1	1	10	rd	0110011	VFNE.R.H	not equal, R

10	10010	rs2	rs1	0	10	rd	0110011	VFLT.H	less than
10	10010	rs2	rs1	1	10	rd	0110011	VFLT.R.H	less than, R
10	10011	rs2	rs1	0	10	rd	0110011	VFGE.H	greater than or equal
10	10011	rs2	rs1	1	10	rd	0110011	VFGE.R.H	greater than or equal, R
10	10100	rs2	rs1	0	10	rd	0110011	VFLE.H	less than or equal
10	10100	rs2	rs1	1	10	rd	0110011	VFLE.R.H	less than or equal, R
10	10101	rs2	rs1	0	10	rd	0110011	VFGT.H	greater than
10	10101	rs2	rs1	1	10	rd	0110011	VFGT.R.H	greater than, R
10	11000	rs2	rs1	0	10	rd	0110011	VFCPKA.H.S	2xbinary32 → binary16 *op0,1*

Xfvec Vectorial Floating-Point Ext. with Xf8, FLEN≥16, vfmt=11 (binary8)

10	00001	rs2	rs1	0	11	rd	0110011	VFADD.B	$rs1 + rs2$
10	00001	rs2	rs1	1	11	rd	0110011	VFADD.R.B	$rs1 + rs2$, R
10	00010	rs2	rs1	0	11	rd	0110011	VFSUB.B	$rs1 - rs2$
10	00010	rs2	rs1	1	11	rd	0110011	VFSUB.R.B	$rs1 - rs2$, R
10	00011	rs2	rs1	0	11	rd	0110011	VFMUL.B	$rs1 * rs2$
10	00011	rs2	rs1	1	11	rd	0110011	VFMUL.R.B	$rs1 * rs2$, R
10	00100	rs2	rs1	0	11	rd	0110011	VFDIV.B	$rs1/rs2$
10	00100	rs2	rs1	1	11	rd	0110011	VFDIV.R.B	$rs1/rs2$, R
10	00101	rs2	rs1	0	11	rd	0110011	VFMIN.B	min
10	00101	rs2	rs1	1	11	rd	0110011	VFMIN.R.B	min, R
10	00110	rs2	rs1	0	11	rd	0110011	VFMAX.B	max
10	00110	rs2	rs1	1	11	rd	0110011	VFMAX.R.B	max, R
10	10111	00000	rs1	0	11	rd	0110011	VFSQRT.B	$\sqrt{rs1}$
10	01000	rs2	rs1	0	11	rd	0110011	VFMAC.B	$(rs1 * rs2) + rd$
10	01000	rs2	rs1	1	11	rd	0110011	VFMAC.R.B	$(rs1 * rs2) + rd$, R
10	01001	rs2	rs1	0	11	rd	0110011	VFMRE.B	$(rs1 * rs2) - rd$
10	01001	rs2	rs1	1	11	rd	0110011	VFMRE.R.B	$(rs1 * rs2) - rd$, R
10	01101	rs2	rs1	0	11	rd	0110011	VFSGNJ.B	rs1, sign of rs2
10	01101	rs2	rs1	1	11	rd	0110011	VFSGNJ.R.B	rs1, sign of rs2, R
10	01110	rs2	rs1	0	11	rd	0110011	VFSGNJN.B	rs1, inv. sign of rs2
10	01110	rs2	rs1	1	11	rd	0110011	VFSGNJN.R.B	rs1, inv. sign of rs2, R
10	01111	rs2	rs1	1	11	rd	0110011	VFSGNJX.B	rs1, sign rs1 ⊕ sign rs2
10	01111	rs2	rs1	0	11	rd	0110011	VFSGNJX.R.B	rs1, sign rs1 ⊕ sign rs2, R
10	10000	rs2	rs1	0	11	rd	0110011	VFEQ.B	equal
10	10000	rs2	rs1	1	11	rd	0110011	VFEQ.R.B	equal, R
10	10001	rs2	rs1	0	11	rd	0110011	VFNE.B	not equal
10	10001	rs2	rs1	1	11	rd	0110011	VFNE.R.B	not equal, R
10	10010	rs2	rs1	0	11	rd	0110011	VFLT.B	less than
10	10010	rs2	rs1	1	11	rd	0110011	VFLT.R.B	less than, R
10	10011	rs2	rs1	0	11	rd	0110011	VFGE.B	greater than or equal
10	10011	rs2	rs1	1	11	rd	0110011	VFGE.R.B	greater than or equal, R
10	10100	rs2	rs1	0	11	rd	0110011	VFLE.B	less than or equal
10	10100	rs2	rs1	1	11	rd	0110011	VFLE.R.B	less than or equal, R
10	10101	rs2	rs1	0	11	rd	0110011	VFGT.B	greater than
10	10101	rs2	rs1	1	11	rd	0110011	VFGT.R.B	greater than, R

Unless RV32D Supported

10	01100	00001	rs1	0	11	rd	0110011	VFCLASS.B	classify
10	01100	00010	rs1	0	11	rd	0110011	VFCVT.X.B	to vector of sgn. bytes
10	01100	00010	rs1	1	11	rd	0110011	VFCVT.XU.B	to vector of usgn. bytes
10	01100	00011	rs1	0	11	rd	0110011	VFCVT.B.X	from vector of sgn. bytes
10	01100	00011	rs1	1	11	rd	0110011	VFCVT.B.XU	from vector of usgn. bytes

Conversions when F Standard Extension Supported

10	11000	rs2	rs1	0	11	rd	0110011	VFCPKA.B.S	2xbinary32 → binary8 *op0,1*
10	11000	rs2	rs1	1	11	rd	0110011	VFCPKB.B.S	2xbinary32 → binary8 *op2,3*

Conversions when Xf16 Extension Supported

10	01100	00111	rs1	0	10	rd	0110011	VFCVT.H.B	*op0,1* binary8 → binary16
10	01100	00111	rs1	1	10	rd	0110011	VFCVTU.H.B	*op2,3* binary8 → binary16
10	01100	00110	rs1	0	11	rd	0110011	VFCVT.B.H	binary16 → binary8 *op0,1*
10	01100	00110	rs1	1	11	rd	0110011	VFCVTU.B.H	binary16 → binary8 *op2,3*

When Xfvec Extension Supported, FLEN≥32, vfmt=10 (binary16)

10	01011	rs2	rs1	0	10	rd	0110011	VFDOTPEX.S.H	fp32(dotp(rs1,rs2))
10	01011	rs2	rs1	1	10	rd	0110011	VFDOTPEX.S.R.H	fp32(dotp(rs1,rs2)), R
10	01011	rs2	rs1	0	10	rd	0110011	VFNDOTPEX.S.H	fp32(ndotp(rs1,rs2))
10	01011	rs2	rs1	1	10	rd	0110011	VFNDOTPEX.S.R.H	fp32(ndotp(rs1,rs2)), R
10	0011110110		rs1	0	00	rd	0110011	VFSUMEX.S.H	fp32(sum(rs1,rd))
10	1011110110		rs1	0	00	rd	0110011	VFNSUMEX.S.H	fp32(sum(rs1,rd))

When Xfvec Extension Supported, FLEN≥32, vfmt=10 (binary16)

10	01011	rs2	rs1	0	00	rd	1110111	FCDOTPEX.S.H	cdotp(rs1,rs2)
10	01011	rs2	rs1	1	00	rd	1110111	FCNDOTPEX.S.R.H	cndotp(rs1,rs2), R
10	11101	rs2	rs1	0	00	rd	1110111	FCCDOTPEX.S.H	cdotp(rs1,rs2*)
10	11101	rs2	rs1	1	00	rd	1110111	FCCNDOTPEX.S.R.H	cndotp(rs1,rs2*), R

When Xfvec Extension Supported, FLEN≥16, vfmt=11 (binary8)

10	01011	rs2	rs1	0	11	rd	0110011	VFDOTPEXA.S.B	fp32(dotp(rs1,rs2))
10	01011	rs2	rs1	1	11	rd	0110011	VFDOTPEXA.S.R.B	fp32(dotp(rs1,rs2)), R
10	01011	rs2	rs1	0	11	rd	0110011	VFDOTPEXB.S.B	fp32(dotp(rs1,rs2))
10	01011	rs2	rs1	1	11	rd	0110011	VFDOTPEXB.S.R.B	fp32(dotp(rs1,rs2)), R
10	0011110111		rs1	0	10	rd	0110011	VFSUMEX.H.B	fp16(sum(rs1,rd))
10	1011110111		rs1	0	10	rd	0110011	VFNSUMEX.H.B	fp16(sum(rs1,rd))

B Vector instructions

The *Xpulpimg* extensions are a subset of the *RVV-1.0* extensions. The *xpulpv2* extensions are documented in `https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf`. The subset implemented in TeraPool's IPU is:

- **SCALAR arithmetic**:
  - P.ABS, P.SLET, P.SLETU,
  - P.MIN, P.MINU, P.MAX, P.MAXU,
  - P.EXTHS, P.EXTHZ, P.EXTBS, P.EXTBZ,
  - P.CLIP, P.CLIPU, P.CLIPR, P.CLIPUR,
  - P.MAC, P.MSU
- **SIMD2 arithmetic:** These instructions are available also in the .SC and .SCI format, where the second term of the computation is either the lower half of the second input register or an immediate.
  - PV.ADD.H, PV.SUB.H,
  - PV.AVG.H, PV.AVGU.H, PV.MIN.H, PV.MINU.H, PV.MAX.H, PV.MAXU.H,
  - PV.SRL.H, PV.SRA.H, PV.SLL.H,
  - PV.OR.H, PV.XOR.H, PV.AND.H, PV.ABS.H,
  - PV.DOTUP.H, PV.DOTUSP.H, PV.DOTSP.H, PV.SDOTUP.H, PV.SDOTUSP.H, PV.SDOTSP.H
- **SIMD4 arithmetic:** These instructions are available also in the .SC and .SCI format, where the second term of the computation is either the lower half of the second input register or an immediate.
  - PV.ADD.B, PV.SUB.B,
  - PV.AVG.B, PV.AVGU.B, PV.MIN.B, PV.MINU.B, PV.MAX.B, PV.MAXU.B,
  - PV.SRL.B, PV.SRA.B, PV.SLL.B,
  - PV.OR.B, PV.XOR.B, PV.AND.B, PV.ABS.B,
  - PV.DOTUP.B, PV.DOTUSP.B, PV.DOTSP.B, PV.SDOTUP.B, PV.SDOTUSP.B, PV.SDOTSP.B.
- **SIMD shuffle:**
  - PV.EXTRACT.H, PV.EXTRACT.B, PV.EXTRACTU.H, PV.EXTRACTU.B,
  - PV.INSERT.H, PV.INSERT.B, PV.SHUFFLE2.H, PV.SHUFFLE2.B,
  - PV.PACK, PV.PACK.H

Additionally, in the Snitch core, we also support the post-increment load and store instructions of the *xpulpv2* extension: P.LBU.IRPOST, P.LH.IRPOST, P.LHU.IRPOST, P.LW.IRPOST, P.LB.RRPOST, P.LBU.RRPOST, P.LH.RRPOST, P.LHU.RRPOST, P.LW.RRPOST, P.LB.RR, P.LBU.RR, P.LH.RR, P.LHU.RR, P.LW.RR, P.SB.IRPOST, P.SH.IRPOST, P.SW.IRPOST, P.SB.RRPOST, P.SH.RRPOST, P.SW.RRPOST, P.SB.RR, P.SH.RR, P.SW.RR.