MemPool: A Scalable Manycore Architecture with a Low-Latency Shared L1 Memory

Samuel Riedel, Matheus Cavalcante, Renzo Andri, and Luca Benini

Abstract— Shared L1 memory clusters are a common architectural pattern (e.g., in GPGPUs) for building efficient and flexible multi-processing-element (PE) engines. However, it is a common belief that these tightly-coupled clusters would not scale beyond a few tens of PEs. In this work, we tackle scaling shared L1 clusters to hundreds of PEs while supporting a flexible and productive programming model and maintaining high efficiency. We present MemPool, a manycore system with 256 RV32IMAXpulpimg "Snitch" cores featuring application-tunable functional units. We designed and implemented an efficient low-latency PE to L1-memory interconnect, an optimized instruction path to ensure each PE's independent execution, and a powerful DMA engine and system interconnect to stream data in and out. MemPool is easy to program, with all the cores sharing a global view of a large, multi-banked, L1 scratchpad memory, accessible within at most five cycles in the absence of conflicts. We provide multiple runtimes to program MemPool at different abstraction levels and illustrate its versatility with a wide set of applications. MemPool runs at 600 MHz (60 gate delays) in typical conditions (TT/0.80 V/25 °C) in 22 nm FDX technology and achieves a performance of up to 229 GOPS or 192 GOPS/W with less than 2% of execution stalls.

Index Terms—Manycore, RISC-V, general-purpose

1 Introduction

Multicore systems are essential to efficiently run today's compute-intensive, extremely parallel workloads ranging from genomics over computational photography and machine learning to graph processing [1], [2]. The pursuit of energy-efficient parallel processing has led to a continuous increase in core count in modern computer architectures [3]. For example, NVIDIA's H100 Tensor Core graphics processing units (GPUs) feature 144 streaming multiprocessors (SMs) with tens of processing elements (PEs) each [4], while machine learning accelerators like Cerebras' Wafer-Scale Engine boast up to 850 000 cores [5].

Multi-core systems span a wide spectrum from general-purpose to highly domain-specific. In the domain-specific corner, we find specialized systolic-array-based architectures, such as Google's Pixel Visual Core [6] or Tensor Processing Unit [7]. They scale to hundreds of cores, thanks to their neighbor-to-neighbor communication. While those systems are well suited for their target kernels, their rigid, specialized interconnect and execution scheme restrict their application domain. Moreover, their programming model is notoriously difficult to manage. In contrast, general-purpose processors such as Apple's M1 Ultra [8], Intel's Core-i9 [9], and Ampere's Altra [10] combine a few high-performance cores sharing large caches. Their complex cores and caches are versatile and facilitate programming but impose a power cost and limit scalability beyond a few tens of cores.

- *Samuel Riedel and Matheus Cavalcante are with the Integrated Systems Laboratory (IIS), Swiss Federal Institute of Technology, 8092 Zurich, Switzerland. E-mail: {sriedel, matheus}@iis.ee.ethz.ch*
- *Renzo Andri. E-mail: info@renzo.ch*
- *Luca Benini is with the Integrated Systems Laboratory (IIS), Swiss Federal Institute of Technology, Zurich, Switzerland, and also with the Department of Electrical, Electronic and Information Engineering (DEI), University of Bologna, 40126 Bologna, Italy. E-mail: lbenini@iis.ee.ethz.ch.*

A typical architectural pattern that aims for an optimal compromise between efficiency and flexibility is a tightly coupled cluster of simple cores sharing an L1 data memory through a low-latency interconnect. We can find instances of this architectural pattern across numerous domains, from SMs of GPUs [4], to the ultra-low-power GreenWaves' GAP9 processor [11], to the high-performance embedded signal processing with the Kalray processor clusters [12], to aerospace applications with the Ramon Chips' RC64 system [13]. The compute cluster approach can efficiently solve a wide range of today's computing problems. However, it is commonly understood that scaling such tightly coupled clusters to more than a few tens of cores is unfeasible due to the low-latency access requirements to the L1 memory, the aggregate instruction stream's bandwidth of all PEs, and the increasing complexity of interconnects.

Computer architects have devised ways to push performance for tightly coupled clusters. For example, SMs in general-purpose computing on a GPUs (GPGPUs) implement a single instruction, multiple thread (SIMT) scheme, replicating functional units that work in lock-step and sacrifice instruction stream diversity across individual cores. A different approach to scaling beyond tens of cores is replicating the simple compute cluster. This approach is found in GPUs as well as high-performance accelerators such as Manticore [14] or Esperanto's ET-SoC-1 [15]. However, multiple clusters with their L1 memories expand the memory hierarchy and introduce communication overhead between them. Furthermore, the additional hierarchy complicates parallelization and the programming model.

This paper tackles the challenge of building an energy-efficient, general-purpose manycore system by scaling the versatile single cluster architecture to hundreds of cores. We set out to design a manycore system that avoids the problems related to SIMT or multi-cluster designs. Our system has independent and individually programmable, efficient, small

cores with low-latency access to a globally shared L1 memory. With efficient synchronization techniques and fast direct memory access (DMA) engines to move data in and out of the cluster's L1 memory, we aim to design a general-purpose manycore system with a streamlined programming model.

We present MemPool, an open-source, parametric, and flexible manycore architecture scalable to hundreds of independently programmable 32-bit RISC-V cores sharing a large L1 memory. Specifically, we focus on a maximal MemPool configuration with 256 cores and 1 MiB of L1 scratchpad memory (SPM) accessible in at most five cycles in absence of contention. By leveraging small cores designed to tolerate outstanding memory instructions, we can hide L1 interconnect latency and close the gap with respect to the performance of an ideally scaled, physically not implementable, single-cycle access L1 cluster. A highly optimized, scalable, cache-based instruction path supplies cores with instructions, while a DMA engine custom-designed for MemPool efficiently moves data in and out of the L1 memory.

With compiler and runtime support to take full advantage of MemPool's architecture, we present a single-cluster manycore architecture that can easily be programmed and efficiently runs not only data-intensive image processing algorithms with regular and predictable L1 memory access patterns but also workloads like graph processing which are characterized by irregular and unpredictable L1 accesses.

This paper extends our earlier work [16] and presents the following contributions:

- The MemPool architecture; a flexible manycore architecture with shared L1 memory (Section 2). Specifically:
  - A physical-aware design of the low-latency scalable L1 data memory interconnect combined with a lightweight and transparent memory addressing scheme that keeps the memory region which is most often accessed by a core in the same memory bank or close by with minimal access latency and energy consumption (Section 3);
  - An instruction cache architecture highly optimized for energy-efficiency (Section 4);
  - A scalable hierarchical system interconnect, containing a cache hierarchy and a specialized distributed DMA engine to feed the full MemPool cluster with instructions and data (Section 5);
- The full physical implementation and performance, power, and area analysis of MemPool in GlobalFoundries 22FDX fully depleted silicon-on-insulator (FD-SOI) technology node (Section 6);
- Compiler support and software runtimes to efficiently program MemPool, including an implementation of the OpenMP standard and support for a domain-specific language (DSL), namely Halide [17], to program MemPool with high-level abstractions (Section 7);
- An evaluation of MemPool's performance and architectural bottlenecks with a wide set of digital signal processing (DSP) kernels and full-blown applications from a wide range of domains (Section 8);
- The open-source release of MemPool, including open simulator support and the full software infrastructure[1].

1. Available on *https://github.com/pulp-platform/mempool*.

In GlobalFoundries 22FDX technology, MemPool runs at 600 MHz, approximately 60 gate delays, in typical operating conditions (TT/0.80 V/25 °C). MemPool's L1 interconnect keeps the average latency at fewer than six cycles, even for a heavy interconnect load of 0.33 request/core/cycle. Our hybrid addressing scheme helps to reduce the average latency and power consumption further. We demonstrate energy savings up to 28% through various optimizations to the instruction cache with respect to our previous work [16]. Furthermore, we illustrate how to efficiently refill all instruction caches through an Advanced eXtensible Interface (AXI) tree interconnect enhanced with software-controlled caches. MemPool's AXI interconnect offers a bandwidth of 256 B/cycle, which is fully utilized by our distributed DMA. We developed compiler support in GCC and LLVM, aiming to hide architectural latencies to L1 memory through instruction scheduling. Thanks to our OpenMP runtime and Halide framework, the implementation of complex applications is straightforward. MemPool achieves a performance up to 229 GOPS and an energy efficiency up to 192 GOPS/W. On average, the memory system only causes stalls 4% of the execution time, and we achieve 0.96 IPC on MemPool's PEs.

The remainder of this paper is structured as follows. It starts with an overview of MemPool's general architecture and processor design in Section 2. Next, we explain individual components and justify their configuration: Section 3 describes MemPool's L1 data interconnect, and the associated hybrid addressing scheme; Section 4 describes MemPool's instruction cache and its optimization steps; finally, we describe the system interconnect in Section 5, including the instruction cache hierarchy and the DMA engine. We present MemPool's physical implementation in Section 6 before defining its programming model in Section 7 and evaluate its performance in Section 8. Finally, related work and conclusion are discussed in Sections 9 and 10.

2 MemPool Architecture Overview

MemPool is a flexible and parametric manycore architecture with hundreds of individually programmable cores sharing a low-latency L1 SPM. Its architecture is shown in Fig. 1 and detailed in a bottom-up fashion in the subsequent sections.

We use the Snitch core [18] as PE, a small single-stage 32-bit processor implementing RISC-V's RV32IMAXpulpimg instruction set architecture (ISA). A distinctive feature of Snitch is the capability to handle multiple outstanding instructions: this is key to tolerating the latency of load and stores operation in MemPool. In Section 2.1, we describe our modifications to Snitch and its key features in detail.

The first building block of MemPool's hierarchy is the *tile*. It combines a few cores, the first levels of the instruction cache, and a subset of the shared L1 memory banks connected with a fully-connected crossbar to the cores. Each tile also has a parameterizable number of *remote* ports, which allows the cores to make requests into remote tiles' SPM. Analogously, each tile has incoming request ports to serve the memory requests from remote tiles. Those ports connect directly to the fully-connected crossbar inside the tile. Incoming and outgoing request ports can optionally be pipelined to reduce the critical path at the cost of latency. Section 3 explains the PE to L1 SPM interconnect in more detail. Every tile has its

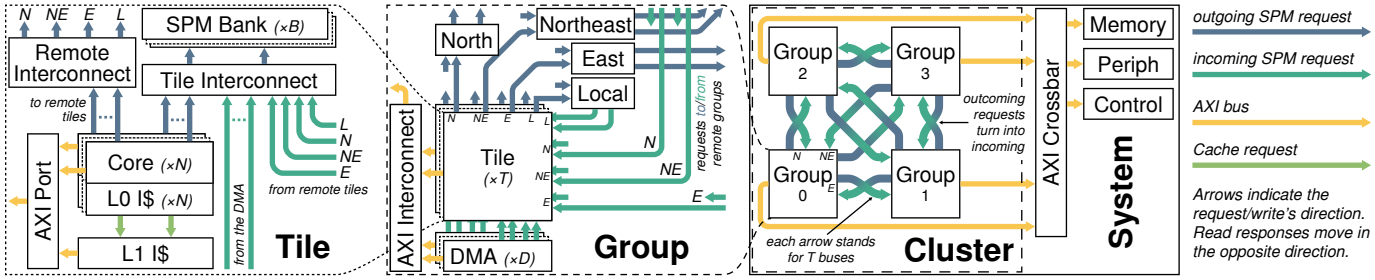


Fig. 1. Bottom-up overview of MemPool's architecture highlighting its hierarchy and interconnects. From left to right, it starts with the *tile*, which holds N cores with private L0 and a shared L1 instruction cache, B SPM banks, and remote connections. The *group* features T such tiles and a *local* L1 interconnect to connect tiles within the group, as well as a *north*, *northeast*, and *east* L1 interconnect connecting to other groups. It also holds a hierarchical AXI interconnect and DMAs. The cluster contains four groups and connects to the system's memory and peripherals through an AXI crossbar. This paper focuses mainly on a large MemPool configuration with $T = 16$ tiles per group with $N = 4$ cores and $B = 16$ banks each.

own instruction cache consisting of a shared L1 cache and small, tightly-coupled L0 caches that are private to each core. Section 4 explains the caches in more detail. The tile also features an AXI port connecting the cores and the cache refill to a system bus to access memory higher up in the hierarchy.

Multiple tiles are combined and connected in a *group*. Each tile's remote port connects to one of several interconnects, which routes the requests to the target group and its tile's SPM. The group also contains a tree-like AXI interconnect and instruction cache hierarchy to connect to the system bus. The AXI connection to the system, including its cache hierarchy and DMA, are further described in Section 5.

Finally, several groups compose a MemPool *cluster* with point-to-point SPM interconnect connections. The cluster also forwards the groups' AXI interconnect to the system-on-chip (SoC). On a system level, MemPool connects to a long-latency (tens to hundreds of cycles) L2 memory, e.g., a large on-chip background memory or an off-chip dynamic random-access memory (DRAM). It also requires access to system-level control registers.

2.1 Core

The PEs of MemPool are RISC-V cores. To scale the architecture, they must have a small footprint, tolerate and hide the L1 interconnect latency, and be extensible to support custom instructions. Snitch, a single-stage 32-bit RISC-V core supporting the RV32IMAFD instruction set, meets these requirements [18]. While Snitch is a single-issue core, it features a scoreboard supporting multiple outstanding instructions, which is crucial for two reasons. First, it allows issuing multiple outstanding load or store requests without blocking the pipeline if there are no read-after-write (RAW) dependencies. Second, Snitch features an accelerator port to add complex, pipelined functional units such as a multiplier. Snitch can offload suitable instructions to those functional units and continue its operation.

We extend Snitch for MemPool by adding the capability to reorder load responses and retire them out-of-order, which is required due to MemPool's non-uniform memory access (NUMA) interconnect not giving ordering guarantees for responses. By supporting up to eight outstanding transactions, Snitch can completely hide the L1 interconnect's five cycles of latency through instruction scheduling with headroom for further load-induced latency.

We configure Snitch to support RV32IMAXpulpimg. The *Xpulpimg* instruction set [19] is supported by a pipelined image processing unit (IPU) connected to Snitch's accelerator port. The unit's instructions like multiply–accumulate (MAC) and load-post-increment drastically reduce the instruction count for DSP kernels and result in a significant speedup. To fit the new changes, we extend Snitch's register file to have three read ports to supply all the operands necessary for MAC operations, and two write ports to reduce contention when retiring instructions with different latencies[2].

MemPool uses the Snitch cores as compute cores that do not consider external interrupts or context switches, which would complicate retiring instructions out-of-order. Furthermore, the L1 interconnect can not create error responses, simplifying Snitch's out-of-order retiring.

2.2 MemPool Configuration

While the MemPool design is highly parametric, for clarity, we mainly focus on the largest and most challenging configuration from the physical implementation viewpoint. Specifically, this maximum configuration contains 256 cores distributed across four groups. Each group consists of 16 tiles containing four cores each. The whole cluster features 1024 1 KiB static random-access memory (SRAM) banks each, resulting in 1 MiB of SPM and a banking factor or 4. Each tile has 32 instructions per core's private L0 cache and 2 KiB of two-way set-associative, shared L1 instruction cache. We configure the AXI data width to be 512-bit wide and parameterize the hierarchical AXI interconnect to result in one AXI master port per group. Each group contains one 8 KiB read-only (RO) cache. This large MemPool configuration features significantly more PEs and larger L1 memory than most common L1-coupled clusters, which range from 8 PEs and 128 KiB of L1 memory [14] to 64 PEs [13].

3 L1 Memory Interconnect

In this section, we elaborate on how to connect cores to L1 SPM and the resulting hierarchy. We motivate the design choices for our interconnect by exploring three alternative interconnect topologies, enhancing them with a hybrid addressing scheme, and comparing their performance.

2. We focus on Xpulpimg as an exemplary configuration for benchmarking the architecture. Snitch can support a wide range of ISA extensions, including floating point, SIMD, etc.

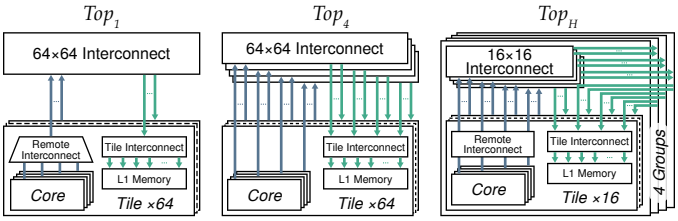


Fig. 2. Evaluated interconnect topologies highlighting the connection between cores and L1 memory in remote tiles.

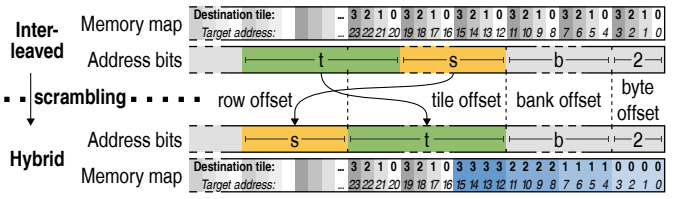


Fig. 3. Address scrambling transforming a fully interleaved memory map (top) to a hybrid one (bottom). The outer bars visualize the memory map with interleaved regions in gray and sequential regions in blue. For simplicity, we assume only four different tiles represented by shades.

Designing a low-latency L1 interconnect that links 256 PEs (initiators) with 1024 memory banks (targets) is a major challenge. High memory throughput, low access latency, and a shared view of memory are required while minimizing the area overhead and ensuring that the interconnect is physically implementable. 1024 1 KiB SRAM banks form the 1 MiB L1. The banking factor of 4 helps to reduce the probability of banking conflicts without requiring the programmer to schedule each core's bank accesses explicitly.

3.1 Interconnect Topologies

The straightforward implementation of the L1 memory interconnect is a fully-connected 256×1024 crossbar. It provides the highest possible throughput thanks to dedicated paths between each master-slave pair eliminating internal routing conflicts. Unfortunately, a crossbar has a quadratic area cost [20], obviously making it prohibitively expensive at MemPool's scale. Instead, we implement this interconnect hierarchically, with *tiles* containing cores and SRAM banks connected at higher hierarchy levels. Fig. 1 shows the tile architecture. It contains four cores and 16 L1 SRAM banks of 1 KiB each, accessible in a single cycle by the tile's cores.

The MemPool cluster contains 64 tiles, totalling 256 cores and 1 MiB of L1 memory. In [16], we proposed and evaluated the three topologies shown in Fig. 2, balancing the design's physical feasibility, access latency, and throughput.

Top_1 has a single outgoing and incoming remote port per tile. The tiles are connected with a 64×64 radix-4 butterfly network, with a single pipeline stage midway through its $\log_4(64) = 3$ layers, resulting in a latency of 5 cycles for remote accesses. Inside the tile, the 5×16 fully-connected *tile interconnect* connects the local cores and incoming port to the 16 banks. The single remote port becomes the bottleneck since four cores compete for access. Top_4 tackles this bottleneck with four remote request ports, one per core. The tiles are connected by four independent 64×64 radix-4 butterfly networks with a core-to-memory latency of 5 cycles.

Top_H also has four outgoing and incoming remote request ports per tile. However, to make this topology feasible, we exploit the physical proximity of the different tiles and add another hierarchy level to MemPool, the *group*. It consists of 16 tiles, a fourth of the MemPool cluster. Cores can access remote data in the same group with a latency of 3 cycles through a 16×16 fully-connected crossbar internal to the group. Moreover, each group pair is connected through an independent 16×16 fully-connected crossbar, with a latency of 5 cycles. Therefore, each group has four 16×16 fully-connected crossbars, one to connect the tiles within the same group (*local* interconnect) and three others to connect to remote groups (*north*, *northeast*, and *east* interconnects).

3.2 Hybrid Addressing Scheme

The L1 interconnect topologies are the first step towards a shared-memory manycore system. However, the opportunity to access remote tiles' memory comes with trade-offs. For all topologies, remote requests have lower throughput, higher latency, and consume more power. We propose a *hybrid addressing scheme*, to mitigate those effects by keeping as many memory requests as possible within a tile.

MemPool has a word-level interleaved memory mapping across all memory banks to minimize banking conflicts. However, this implies that most memory requests target remote tiles. Ideally, most requests remain in the local tile to minimize latency and power consumption. With the scrambling logic of Fig. 3, we transform an interleaved memory map into a *hybrid* one, by creating *sequential regions* (in blue) in which contiguous addresses target a single tile.

In an interleaved memory addressing scheme, the addresses are interpreted as follows. The first two bits are the byte offset, after which b bits identify one of each tile's 2^b banks. The next t bits distinguish between the 2^t tiles. The remaining bits define the row offset within the bank. In a continuous addressing scheme, lower address bits represent the bank's row offset, and the upper bits select a bank. We dedicate 2^s rows of each tile's banks, or 2^{s+b+2} B in each tile, to the sequential memory region. To keep accesses within the same tile interleaved, we leave the byte and bank offsets untouched. The next s bits represented part of the tile offset but should now define the banks' next row within the same tile. Hence, we shift them t bits to the left to the start of the row offset. Incrementing these address bits will consequently traverse the rows of banks within one tile, while the tile offset should stay constant. Therefore, we fill the shifted bits with the t bits they replaced. This permutation creates 2^t sequential regions, one for each tile. In total, we dedicate the first $2^{t+s+b+2}$ B to sequential regions. We leave the subsequent bytes interleaved by only scrambling addresses inside the sequential memory region. The scrambling logic can be efficiently implemented in hardware with a wire crossing and a multiplexer.

The hybrid addressing scheme's key benefit is to allow the programmer to store private data, such as the stack, locally in the PE's tile. This greatly reduces the number of transactions between tiles, making better use of the tiles' local high-throughput L1 crossbar. In contrast to aliased, fully private memories, we do not complicate programmability. By

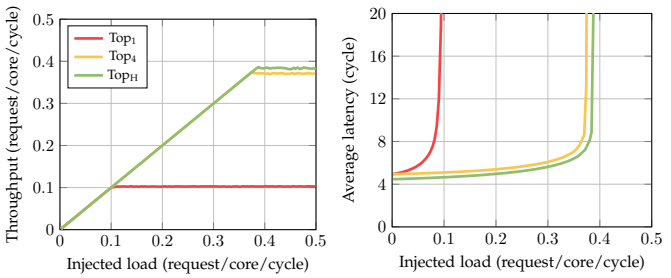


Fig. 4. Network analysis of the three proposed topologies, in terms of throughput and average round-trip latency, as a function of the load.

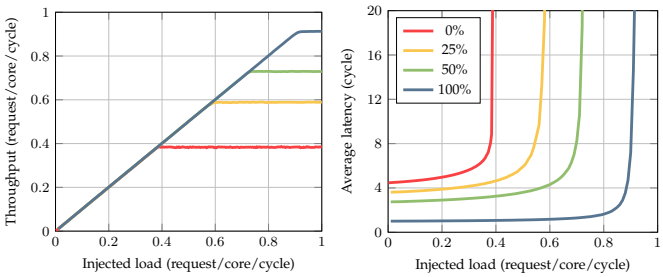


Fig. 5. Network analysis of Top_H with our hybrid addressing scheme for different probabilities of requesting data in the sequential region p_local.

applying the same address transformation for all cores, we give them the same memory view and keep the L1 memory region contiguous and consistent among all PEs. Programs that heavily use the stack or work mainly on local data immensely benefit from the sequential regions.

3.3 Evaluation

3.3.1 L1 Memory Interconnect

We analyze the physical feasibility of the presented topologies presented. We find that both Top_1 and Top_H are feasible, while Top_4 is not, because of the large routing requirements of its four 64×64 global interconnects [16]. Traffic generators, which generate new requests following a Poisson process of rate λ, replace the cores to analyze the topologies' latency and throughput as a function of the injected load λ (measured in requests per core per cycle). The requests have a random, uniformly distributed destination bank.

Section 3.3.1 shows the throughput of different topologies. At an injected load of 0.10 request/core/cycle, Top_1 becomes congested, while Top_4 and Top_H support almost four times that load, about 0.38 request/core/cycle. Top_H's throughput is slightly higher than Top_4's due to its smaller diameter.

Section 3.3.1 shows the average round-trip latency. The explosion of the average latency highlights the point where the topologies become congested. Top_H's average latency only reaches 6 cycles at a network load of 0.33 request/core/cycle. Due to Top_H's three-cycle latency to a local group, it achieves a smaller average latency than Top_4. Since Top_H clearly outperforms Top_1 as well as Top_4, which is physically infeasible, we implement Top_H as MemPool's L1 interconnect.

3.3.2 Hybrid Addressing Scheme

To illustrate the benefit of the hybrid addressing scheme, we analyze Top_H taking the hybrid addressing scheme into account. The traffic generator creates uniformly-distributed requests to the local tile's sequential region with probability p_local, and outside of this region with probability $1 - p_\text{local}$.

Section 3.3.2 shows the throughput of Top_H for different p_local. It shows a clear trend of an increased throughput for a larger p_local. The scrambling logic can vastly improve the system's throughput by preventing congestion in the global interconnect, besides lowering the overall average access latency as seen in Section 3.3.2. An application making 25% of its accesses to the stack can gain up to 50% in performance with the hybrid addressing scheme without changing the code. Due to the clear benefit of using the hybrid addressing scheme, we always enable it in MemPool.

4 Instruction Cache

With the cores connected to the SPM, we must address their instruction path. This section describes the implementation and optimization of the tile's instruction cache.

4.1 Implementation

Since Snitch is a single-stage processor, we must carefully design its instruction cache to minimize the impact on its critical path. There is no instruction fetch stage and, therefore, no pipeline stages in Snitch, meaning the combinational path of the cache providing a requested instruction will be stacked to the path crossing the whole processor. For this reason, each processor has a minimal, private, fully associative, standard cell memory (SCM)-based L0 cache. Its line width and count are configurable. To avoid the miss penalty of such a small cache, it implements prefetching, which requests the next line ahead of time and scans the current cache line for backward branches, representing loops, or a predictable jump.

A shared L1 cache per tile feeds the L0 caches. It implements a configurable set-associative cache lookup where the data and tag banks can be implemented as SRAM macros or SCM. Its refill logic coalesces outstanding refill requests and responds to all L0 caches in parallel to reduce latency. The refill port is connected to the tile's wide AXI port.

We optimize and evaluate the following cache configurations and measure the impact on speed and power with cycle-accurate post-layout power simulations:

Baseline: (149 kGE) The cache architecture used in the preliminary MemPool version [16]. The L0 cache is register-based, and each core has four 128-bit lines. The L1 cache implements a 2 KiB, 4-way set-associative, parallel lookup where the data, as well as the tag banks, are SRAM-based. For a lookup, we read the complete set (all the tag and data banks) simultaneously and then select the correct cache line from the hit calculation.

2-Way: (163 kGE) The cache line width is doubled to 256 bit. The key idea behind this change is to alleviate pressure on the L1 interface. Previously, each of the four L0 caches had to request a new cache line every four instructions executed, fully utilizing the L1 interface unless the code completely fits into the L0 cache. With eight instructions per line, each L0 cache only requires a new cache line every eight instructions, leaving room for mispredicted branches and jumps. We reduce the associativity to two to compensate for the wider SRAM banks in the L1 cache. In combination, this keeps the L1 cache size

constant while doubling the L0 cache, which allows bigger kernels to fit entirely in the L0 cache, reducing instruction stalls and L1 power consumption.

L1-Tag Latch: (161 kGE) This configuration replaces the SRAM-based tag banks of the L1 cache with latch-based SCMs to reduce power and area (due to its small size).

L1-All Latch: (217 kGE) Going a step further, we also replace the L1 cache's data banks with latch-based SCMs. This drastically increases area as the data banks are big enough to benefit from being implemented in SRAM. We discard this solution due to the significant area overhead, which conflicts with our physical constraints.

L1-Tag+L0 Latch: (153 kGE) Rather than making the L1 data latch-based, we replace the registers in the L0 cache with latches for area and power reduction.

Serial L1: (123 kGE) The last architecture implements a serial L1 lookup where first the SCM-based tags are checked in parallel, and then the data from the way that hit is read. This allows coalescing the SRAM banks of both data ways into a single bank, which is more area-efficient than two banks that can be read in parallel and conserves energy by minimizing the number of SRAM accesses. This serial lookup is fully pipelined, keeping the throughput constant, but inherently adds an extra cycle of lookup latency. However, the prefetching of the L0 cache can hide this latency during regular operation.

4.2 Evaluation

To evaluate the cache architectures, we analyze their power consumption using the methodology described in Section 6.1. We use two different workloads taken from our benchmark to give an exemplary quantitative assessment of the different cache organization options: 1) a *small* kernel that fits into the optimized L0 cache and 2) a *big* kernel that never fits into the L0 cache. The *small* kernel has a 2% speedup from doubling the L0 cache size. Since it fits into the L0 cache, there is no performance improvement by changing the SCM type and the final L1 cache optimization. The *big* kernel gains 5% of performance through the first optimization and then loses 3% through the additional latency of the serial lookup. Performance changes are small because the L0 cache's prefetching hides most misses in all configurations. Both kernels achieve higher performance on the final architecture.

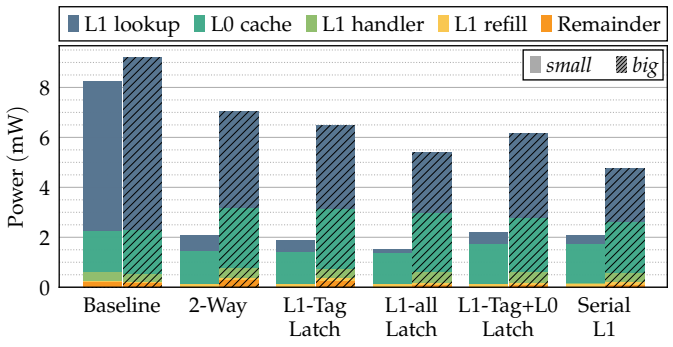


Fig. 6. Breakdown of the instruction cache's power consumption at various optimization steps for a *small* kernel that fits into the L0 cache and a *big* one that does not.

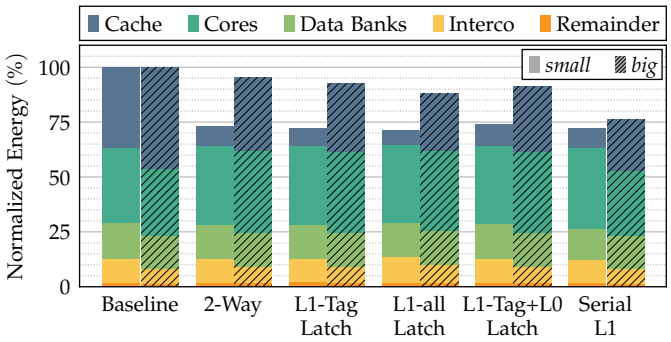


Fig. 7. Breakdown of the tile's normalized energy consumption at various optimization steps for a *small* kernel that fits into the L0 cache and a *big* one that does not.

Figure 6 shows the cache architectures' power consumption. In the baseline version, the power consumption is dominated by the SRAM banks. Increasing the cache size to make the kernel fit into L0 diminishes the L1 lookup power for the *small* kernel, and in combination with the later optimizations, it saves 6.2 mW (−75%). The *big* kernel, which continuously has to be refilled from L1, stresses the benefits of the later optimizations. The L1 cache's power consumption decreases by 45% when reducing the associativity and thereby halving the SRAM reads. Moving to latch-based SCMs reduces power further. Finally, the serial lookup L1 cache architecture reduces its consumption by 36%. Overall, the optimized cache saves 4.5 mW (−48%) per tile.

The tile's energy consumption is shown in Fig. 7. The slight increase in the cores' energy consumption suggests the performance increase thanks to the larger L0 cache. Energy efficiency gains of 28% and 24% for *small* and *big* kernel, respectively, accentuate the key role of an efficient cache subsystem in a fully programmable manycore system.

To summarize, we double the L0 cache size from 16 to 32 instructions by going to a 256-bit-wide cache line; we move from 4-way to 2-way set associativity and go from a parallel to a serial lookup architecture in the L1 cache; we change the L0 cache and the L1 cache's tag to be latch-based while keeping the banks SRAM-based. Overall, we reduce the cache area by 17% and improve the performance of our benchmarks by 2%. We save 5.9 mW (−75%) in one tile when the kernel fits into L0; otherwise, we save 4.4 mW (−48%). For the full MemPool with 64 tiles, this amounts to 378 mW and 282 mW, respectively.

5 System Interconnect and DMA Engine

With the tile's cache being optimized and the cores having access to a shared L1 data memory, we need to make sure the caches can refill and that we can transfer data to and from the system memory. Giving each of the 64 tiles a private high-bandwidth connection to the system is exceptionally challenging in an already routing-dominated design. Note, the preliminary version of MemPool [16] did not consider such an interconnect and instead assumed an ideal connection between the tiles and system memory. This section tackles the challenge of connecting all 64 tiles' AXI port to the system bus with minimal area and routing overhead. Specifically, we need to extend the instruction

cache hierarchy to sustain the cores' instruction stream from L2 or external memory, and we need to design a special DMA that can transfer data between MemPool's distributed L1 memory and L2 or external memory with minimal overhead and without congesting the cores' interconnects.

5.1 Hierarchical AXI interconnect

The instruction and data path are latency tolerant and take advantage of spatial locality by operating on contiguous memory chunks. Therefore, we use the open and standard AXI protocol to connect the tiles to the system, as it is burst-based, latency-tolerant, multi-initiator/target with decoupled read and write channels.

Each tile has an AXI master port shared between all cores and the instruction cache refill, giving them access to higher-level/main memory, peripherals, and control registers. However, routing a private AXI bus per tile to the cluster boundary is not physically feasible. For this reason, we designed the AXI interconnect with a hierarchical approach as shown in Fig. 8. Each tile and DMA is a leaf node in a configurable AXI tree. At each level in the interconnect, neighboring child nodes merge into a single AXI bus. At the top level, we keep a configurable number of master ports to allow for high bandwidth, effectively building multiple AXI trees with a subset of the tiles and DMAs each.

5.2 Read-only cache

Combining multiple AXI masters not only reduces routing but also allows for caches to coalesce requests at each level to reduce the bandwidth requirements. To this end, we implement a specialized RO cache and optionally instantiate it at each node. We leave out write-support to the cache to keep it small and energy-efficient by avoiding coherency problems. The cache is software-managed, giving the programmer full control over cached regions and cache flushes. This enables caching of the binary and RO data during booting and by the programmer.

The RO cache consists of four stages. In the *AXI to cache* stage, AXI bursts are broken down into individual cache requests, and their metadata is stored to reconstruct the correct AXI responses later. The cache requests traverse the *lookup* stage, whose output is processed by the *handler*. In case of a hit, data is returned to the *AXI to cache* stage to issue AXI responses. For a miss, the handler checks if a corresponding refill is in flight or issues a new one. The cache supports multiple outstanding requests. However, the AXI protocol imposes one constraint: requests of the same ID must return in order. Since hits can easily overtake a miss, hits must be stalled if the same master issued a pending miss earlier.

The RO cache's depth and width can be configured and are design parameters that define the bandwidth and the amount of coalescing the cache provides. The cache is designed to be fully pipelined to allow for maximum throughput. We will primarily use the RO cache for instructions since we rarely want to cache DMA transfers or move big chunks of RO data with the processors. Hence, we tune it mainly for the instruction path while also dimensioning the hierarchical AXI interconnect to serve the DMA well.

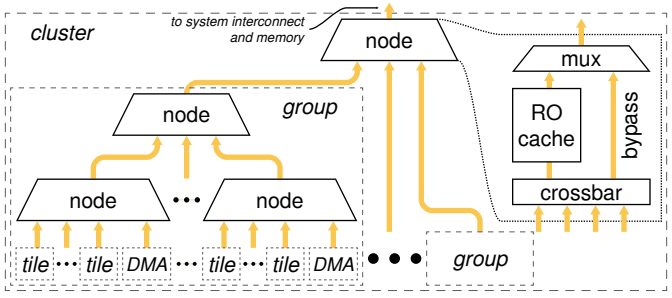


Fig. 8. Hierarchical AXI interconnect with optional read-only caches.

5.3 DMA engine

The final piece of MemPool's architecture is a DMA engine to move data to and from L1 memory efficiently, utilizing the AXI interconnect and having access to all 1024 SPM banks.

Due to MemPool's distributed L1 memory and large scale, a classic DMA engine is not feasible. Since MemPool's L1 memory is shared, there should be a single DMA that can access the entire L1 memory. Having multiple DMAs responsible for exclusive regions complicates programming. However, a single DMA connected to all 1024 memory banks implies excessive extra routing. A possible solution could be to reuse the L1 interconnect. However, this interconnect is not designed for wide data transfers, and a DMA could quickly congest the interconnect, slowing down the cores' transactions. In contrast, the AXI bus connecting all tiles is ideal for DMA type of transfers.

Using a modular DMA engine [21], we design a distributed DMA for MemPool as shown in Fig. 9. It consists of a configurable number of data movers named *backend*, e.g., one per four tiles, which are responsible for their tiles' memory region and connect to the tiles' AXI port on one side and their fully-connected local crossbar on the other. A single configuration *frontend* controls all backends. Specifically, the programmer can request DMA transfers involving the full MemPool cluster, and the DMA control will split the transfer into multiple smaller transfers and coordinate the data movers. To this end, we implement two key modules, a *splitter* and a *distributor*. Both take a DMA request as input and output one or multiple reshaped requests, which the data

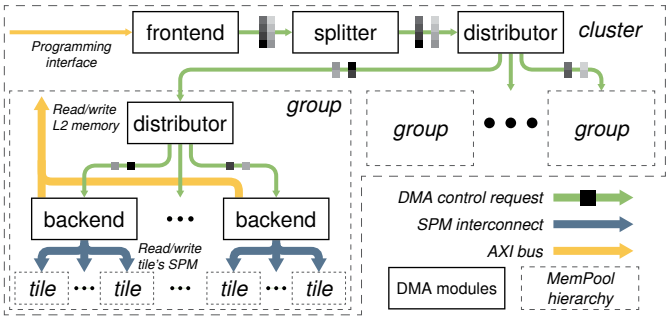


Fig. 9. DMA architecture and its integration in MemPool. The DMA is programmed through a single *frontend* which forwards a full DMA request to the *splitter* module, where the request is split into multiple serial aligned requests. Through a tree of distributor modules, the requests are split into parallel, narrower requests, handled by distinct *backends* that do the data movement. Each backend has access to L2 and to specific tiles.

mover eventually executes. The former splits the transfer at the address boundary that spans one line of MemPool's L1 memory into multiple serial requests, thereby taking the interleaved addressing scheme into account. The latter distributes those requests across multiple parallel requests requiring distinct L1 memory regions. This design has two main benefits. It fully reuses the hierarchical AXI interconnect to bring the data to the tiles with minimal extra routing. Also, the DMAs are connected to the tiles' internal, fully-connected crossbars, as shown in Fig. 1, where the bandwidth is high, and interference with the cores has the least impact.

5.4 System

We connect the AXI ports to a SoC to get access to L2 (or main memory) as well as peripherals. MemPool also requires a few control registers for its runtime, which the SoC could integrate or be merged into the MemPool cluster itself. They allow waking up the cores, hold a few system parameters, such as the core count, and the RO cache configuration.

We evaluate MemPool with a system containing a 128 GiB/s L2 memory for instructions and data, the control registers to configure MemPool and send wake-up pulses, and a boot ROM. Each group has one 512 bit wide AXI bus connecting to the SoC with an access latency of 12 cycles.

5.5 Evaluation

We measure the execution time with a cold cache to evaluate the performance benefit of different AXI interconnect architectures and RO cache widths on the instruction path and compare it with a non-hierarchical, cacheless interconnect. The RO cache performs best if its line is larger or equal to the tiles' cache line width. A radix of eight gives the best performance with a speedup of 1.59×. However, the associated hardware cost of three RO caches is not justified compared to a radix-16 interconnect with only one RO cache and a speedup of 1.54×. Since the DMA's performance is independent of the radix, we opt for the radix-16 solution.

To find the optimal configuration of the DMA, we measure the utilization of MemPool's AXI master ports for various configurations and transfer sizes, as shown in Fig. 10. Up to a certain size, the number of DMA backends makes little difference, and they can fully utilize the system bandwidth for large transfers. Even for very small transfers, the utilization reaches roughly 53%. Using 16 backends, one per tile, drastically reduces performance because each backend is only responsible for the 512 bit of continuous

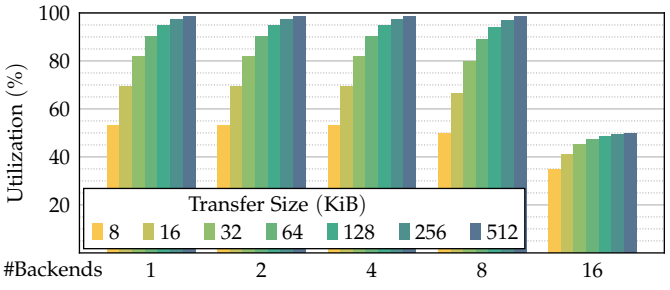


Fig. 10. System bus utilization with different numbers of DMA *Backends* per group for various transfer sizes.

memory in a single tile preventing it from using AXI bursts. Our final implementation uses four DMA backends per group, as this configuration gives the best performance.

6 Implementation

We have explored and evaluated all the critical components of MemPool individually. In this section, we analyze the full MemPool design, describing the implementation methodology, MemPool's final parameterization, and its physical implementation used for the final evaluation.

6.1 Methodology

MemPool targets running at 500 MHz in worst-case conditions (SS/0.72 V/125 °C) when implemented in GlobalFoundries' 22FDX FD-SOI technology. We implement MemPool using Synopsys' Fusion Compiler 2022.03. Due to the size of the design, we adopt a bottom-up implementation flow, first implementing the groups before integrating them as macros into the cluster design. We use Synopsys' PrimeTime 2022.03 to estimate MemPool's power consumption in typical conditions (TT/0.80 V/25 °C), with switching activities extracted from a post-layout gate-level simulation running at 600 MHz. All performance measurements come from cycle-accurate register-transfer level (RTL) simulation.

6.2 Implementation flow

The routing of MemPool is one of the fundamental challenges for its physical implementation. The problem is amplified by MemPool's large size, which demands a hierarchical implementation flow. A key advantage of the chosen Top_H interconnect is the introduction of the additional group hierarchy. It allows us to skip the tile hierarchy, implement a flat group as the first hierarchy level, and later construct the cluster out of group macros. Not creating a tile macro but directly implementing a flat group as the first hierarchy block allows the tools to implement all interconnects flat and seamlessly route through the tiles, which would otherwise be black boxes. This gain in freedom significantly improves resource utilization and allows implementing MemPool on a smaller footprint than in previous work [16]. Despite the addition of a custom DSP unit per core and a full AXI interconnect, including a cache hierarchy and DMA, we can still achieve the same worst-case frequency of 482 MHz while reducing the area by a factor of 1.6 to 12.9 mm^2, thanks to the optimized cache and the improved implementation flow. The deepest path traverses 35 gates in the IPU. However, the interconnect connecting two groups actually limits MemPool's frequency. Its critical path is 40% wire delay and 60% gate delay coming from 28 gates, half of which are buffers.

6.3 Floorplan

The annotated die shot of the fully placed and routed MemPool group can be seen in Fig. 11. We place the macros of the tile memories in a 4×4 grid, flipping the upper half of the grid to create an open area in the middle of the group, where the tool has space to place the interconnects. The annotations show how all the tiles are clustered around their memory

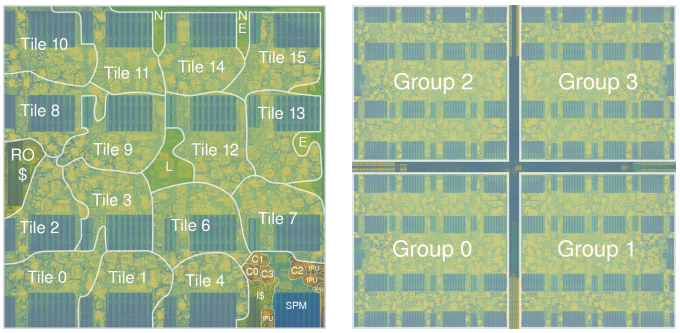


Fig. 11. Annotated dieshot of placed and routed MemPool group (left) and cluster (right). We highlight the tiles, the RO cache, and the interconnects between tiles of the same group (L) and to other groups (N, NE, E) as well as Tile 5's cores, IPU's, instruction cache, and SPM with its interconnect.

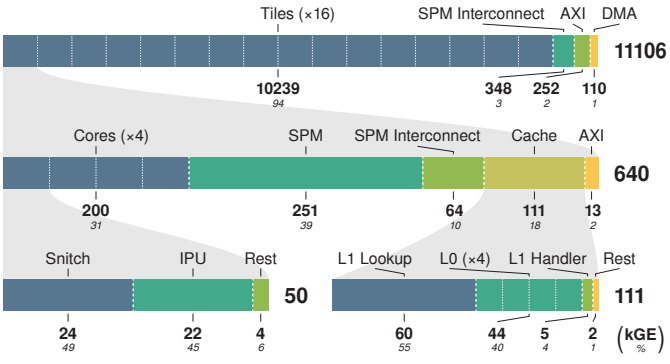


Fig. 12. Hierarchical area breakdown of a MemPool group in **kGE** with annotations showing the *percentage* of the immediate parent component. The whole group occupies 11 MGE, most of which is occupied by tiles, while the interconnects and DMAs only make up a small percentage. Within a tile, the cores and SPM banks make up most of the area.

macros but drawn to the center to minimize wiring delay towards the interconnect. The remote SPM interconnects are drawn to their pins and fit between the tiles. The AXI interconnect and RO cache are placed on the left boundary.

Fig. 12 shows the hierarchical area distribution of MemPool's group. It shows that we succeeded in adding an AXI interconnect with minimal area overhead, despite including a full level of cache hierarchy and a specialized DMA. Within a tile, the SPM banks are the biggest components, followed by the four cores and their instruction cache. The core's area is split between the Snitch core itself and its IPU accelerator.

7 Programming Model

A streamlined and efficient programming model is vital for adopting a new computing platform. We have designed MemPool with programmability as one of its primary goals.[3]

MemPool's shared memory makes programming already much more straightforward than a design with isolated memory regions, such as multi-cluster or systolic architectures. It allows for all-to-all communication and synchronization without considering the memory map. Furthermore, since each core runs independently and can be programmed individually, there is no need to worry about lock-step execution, branch divergence, or thread warping.

3. To facilitate developing applications for MemPool, we support emulating it on Banshee, a binary translation based emulation platform [22].

7.1 Compiler support

We have extended the GNU GCC and the LLVM toolchains to support MemPool's ISA extensions and instruction scheduling. Particularly, both toolchains are aware of MemPool's architectural latency of each instruction and will schedule instructions to prevent RAW hazards in Snitch. Primarily, this entails scheduling load operations as far as possible from their usage in data processing instructions to hide the L1 latency. Other sources of RAW stalls are pipelined instruction offloaded to Snitch's accelerator. Offloading this work to the compiler allows hiding the non-idealities of the L1 interconnect and Snitch's accelerator. To the programmer, MemPool becomes an idealized single-cycle latency cluster.

7.2 Synchronization

A key aspect in programming a manycore system is synchronization. We rely on the RISC-V standard atomic extension for synchronization primitives. Specifically, each L1 memory bank's controller is extended with a small arithmetic logic unit (ALU) to support RISC-V's atomic memory operations (AMOs) and load-reserved/store-conditional (LRSC) instructions. They allow the implementation of mutexes or concurrent algorithms in a standardized fashion. To save energy during synchronization, MemPool also provides sleep and wake-up behavior, where each core can send a wake-up pulse to arbitrary cores or wake up the complete clusters in a single store, enabling very efficient synchronization barriers.

7.3 Runtimes

7.3.1 *Bare-metal runtime*

We provide a bare-metal C runtime for MemPool where cores are programmed with the same code, and the programmer can create branches for each core or let them work on specific data using their unique ID [23]. We hand control over to the programmer once the stack is allocated in the sequential region and the runtime is initialized. The programmer can then use various runtime functions, such as allocators and barriers, and has complete control over every core to implement highly optimized applications. However, implementing large applications becomes time-consuming.

7.3.2 *OpenMP*

We implement the standard OpenMP runtime for MemPool, providing a shared-memory fork-join programming model. The program is primarily executed by a single *master* core, which can then fork off to employ the complete cluster. Specifically, we support static and dynamic loop scheduling; parallel sections; synchronization directives such as master, critical, atomic, and barrier; and data sharing, including reductions. OpenMP simplifies parallelizing a workload at the cost of a runtime overhead as evaluated in Section 8.2.2.

7.3.3 *Halide*

Halide is a DSL aimed at image processing and machine learning [17]. Its key feature is decoupling the functional description of an application from its execution, such as tiling, unrolling, or parallelizing, allowing the functional code to be shared across different platforms while tuning the implementation for each specific architecture. Halide is also

TABLE 1
Benchmark and power results obtained from post-layout simulations. An operation corresponds to a 32-bit addition or multiplication.

Kernel	Size	Util (IPC)	Power (W)	Performance (OP/cycle)	(GOPS/W)
matmul	256×256	0.88	1.25	285	136
2dconv	96×1024	0.87	1.19	336	170
dct	192×1024	0.93	1.02	168	99
axpy	98'304	0.76	1.17	90	46
dotp	98'304	0.74	1.21	92	46

capable of optimizations across kernel boundaries, thanks to its functional programming nature. We extend Halide to support MemPool and implement the necessary runtime.

8 Performance Evaluation

8.1 Microarchitecture Benchmarking

We evaluate MemPool's performance, power, and energy efficiency for DSP kernels from a wide range of domains. All kernels operate on 32-bit integers, are parallelized across all of MemPool's cores, and include a final synchronization barrier. Specifically, we consider the following kernels:

matmul: A matrix-matrix multiplication where each core operates on a 4×4 output tile, fully utilizing Snitch's register file to maximize computational intensity, resulting in eight loads per 16 MAC operations.

2dconv: A 2D convolution with a 3×3 kernel. Cores operate on pixels mapped to their tile, leading to local accesses except for pixels at the edges of a tile. The kernel maximizes data reuse by operating on 4×3 tiles.

dct: Computes the 2D discrete cosine transform (DCT) on 8×8 blocks, as used for JPEG compression. Cores work on local blocks and use the stack for intermediate results.

axpy: A key BLAS routine computing $\alpha \cdot \vec{x} + \vec{y}$ with a low computational intensity with two loads and one store for every MAC. It is optimized only to have local accesses.

dotp: The dot product computes the scalar product of two vectors. Like *axpy*, it has a low computational intensity and is parallelized only to have local accesses.

8.1.1 Performance Results

Table 1 shows the performance metrics of the selected kernels. All results were extracted with the methodology described in Section 6.1. The full MemPool cluster consumes roughly 1 W and consistently achieves a high instructions per cycle (IPC), especially for compute-intensive kernels, where we can reach up to 336 OP/cycle thanks to the MAC extension. In terms of energy-efficiency, MemPool reaches 170 GOPS/W.

8.1.2 Scaling behavior

A comparison with a single-core system, which represents an idealized, conflict-free system, evaluates how well MemPool scales, using weak scaling, which means the problem size scales with the system size. Figure 13 shows the speedup of MemPool with and without a full synchronization barrier after the execution. It allows analyzing whether the inherent final synchronization step or conflicts between cores prevent ideal scaling. For compute-intensive kernels such as *matmul*, *2dconv*, and *dct*, the speedup is very close to ideal even considering the final barrier. While *2dconv* and *dct*'s performance, excluding the synchronization, is on par with the single-core system, the *matmul*'s speedup indicates that conflicts lead to performance losses. For kernels with low compute intensity, the final synchronization step becomes noticeable. However, they still achieve 75% of the ideal speedup. Overall, MemPool achieves speedups very close to the ideal, only losing 10% due to synchronization in compute-intensive workloads.

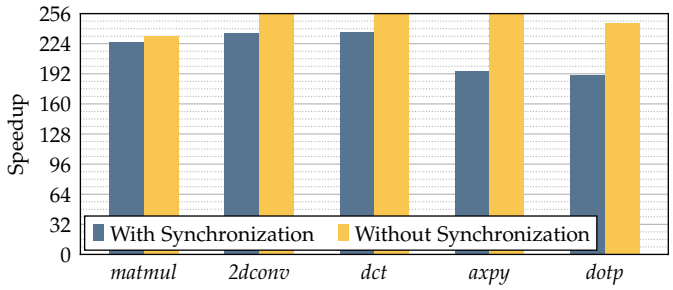


Fig. 13. Speedup of MemPool over single-core execution using weak scaling, i.e., the problem scales with the core count. We show the speedup with and without a final synchronization barrier to analyze the main cause of performance losses.

We investigate the MemPool's non-idealities further by analyzing where the cores spend their time. Some cycles need to be spent on control and memory instructions due to Snitch's simple design and RISC-V's load-store architecture, inhibiting a 100% compute unit utilization. Figure 14 shows the breakdown of cycles spent. It stacks the percentage of compute instructions, representing compute unit utilization, with the percentage of control instructions, totaling in the IPC. Finally, it shows the breakdown of idle cycles due to synchronization or architectural stalls. A clear distinction between compute and memory-intensive kernels can be made out. The former achieve a high compute utilization of up to 66%, while the latter spend more time on load and store instructions inherent to load-store architectures.

Another difference between compute and memory-bound

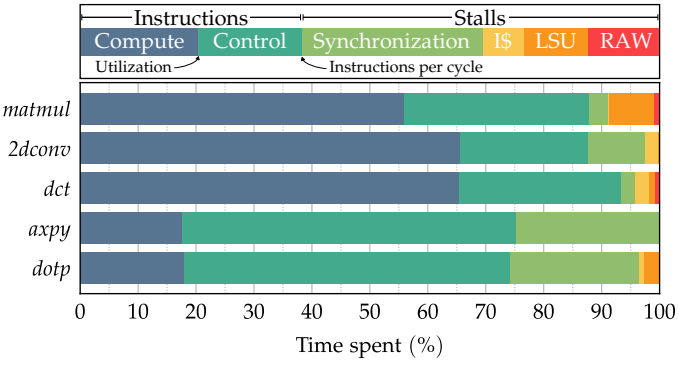


Fig. 14. Breakdown of cores' activity during kernel execution. The first two bars show the time spent executing instructions, differentiating between *compute* and *control* instructions. All other bars indicate idle phases. *Synchronization* represents the time spent sleeping at barriers, *I$* denotes instruction path stalls, *LSU* stalls come from congestion in the interconnect preventing the core from issuing a load or store, and *RAW* stalls denote read-after-write hazards. It contains metrics such as the compute unit utilization, the IPC, and the synchronization costs.

kernels is the synchronization overhead. While the absolute overhead is similar for all kernels, compute-intensive kernels can amortize it by running longer on the same amount of L1 data. Besides synchronization, we observe very few architectural stalls. The *matmul* kernel is the only kernel showing some *LSU* stalls, which come from conflicts in the interconnect, whose increased latency is not fully hidden by instruction scheduling. Thanks to the hybrid addressing scheme, we can keep most of the data locally for the other kernels, resulting in minimal memory conflicts. Only the *dotp* kernel exhibits some memory conflicts for the final reduction step. MemPool's low-latency interconnect and Snitch's ability to manage multiple outstanding memory requests minimizes the RAW stalls to a negligible amount across all kernels. Finally, the well-optimized cache and its prefetching lead to very few instruction interface stalls.

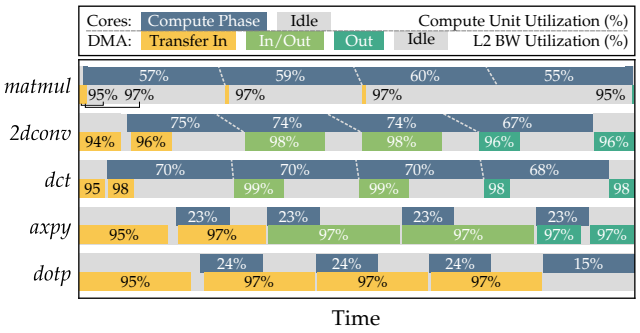


Fig. 15. Timing diagram of double-buffered kernels. The upper bar shows the active computing phases with the compute unit utilization annotated. The lower bars represent the DMA's active phases with the L2 utilization. The top three kernels are compute bound with fused compute phases. Diagonal lines illustrate first PEs moving to the next phase while the last PEs are still working in the previous one.

8.2 System Benchmarking

8.2.1 Double-buffered implementation

To evaluate the performance of the full MemPool, including the system interconnect, we evaluate the kernels on data from system memory, using double-buffering to hide memory transfers. Because the L1 has to buffer two problems at once, the kernels work on half the problem sizes here. The PEs synchronize between rounds, where the first PE that enters a new round checks whether the DMA transfer has finished. The last PE programs the DMA transfer for the next round.

Fig. 15 shows the compute and transfer phases for all kernels, starting with an initial DMA-only phase loading in the first chunk, followed by the first compute phase, which is accompanied by a transfer only bringing data into MemPool. The next two rounds are full compute and transfer rounds, where previous results are stored back to L2 and upcoming inputs transferred in. These stages are the most important as they are replicated for large problem sizes while the ramp-up and down rounds stay consistent. A transfer with only outputs accompanies the last compute round before a final DMA-only phase writes back the last results. If the phases are not fused, it takes roughly 30 cycles to set up a new DMA transfer, and each L2 access has a latency of 12 cycles.

For compute-bound kernels, like *matmul*, *2dconv*, and *dct*, MemPool achieves even higher performance during steady rounds than with a single compute round. It achieves an IPC of 94% or 306 OP/cycle for *matmul*, 0.99 IPC and 178 OP/cycle for *dct*, or even 0.98 IPC and 381 OP/cycle for *2dconv*, which corresponds to 229 GOPS or 192 GOPS/W. This performance increase is mainly due to fusing compute rounds and smaller synchronization overheads. The fused rounds also eliminate a significant portion of the LSU stalls, which mainly originate from cores initially accessing the same banks. The slight drifting of cores and their access patterns leads to more uniformly distributed accesses and fewer conflicts. In the double-buffered execution, the cores can keep this drift between compute phases and suffer fewer LSU stalls. Finally, for heavily memory-bound kernels like *axpy* and *dotp*, the L2 bandwidth is not sufficient to keep all PEs occupied. While they achieve IPCs of 97% and 99%, respectively, the compute phases only last for 35% and 51% of the steady state rounds. Due to the final reduction and writeback, *dotp*'s final compute phase is longer.

8.2.2 OpenMP and Halide applications

We have shown that MemPool can achieve very high performance on well-optimized DSP kernels. While hand-optimized kernels are often used in applications through libraries, we also show that MemPool is conveniently programmable with high-level abstractions such as OpenMP and the DSL Halide with the following applications:

histogram equalization: To enhance an image's contrast, histogram equalization computes a transformation function based on the input image's histogram such that the output image uses the full spectrum of intensities. We implement this algorithm in Halide, where the main challenge lies in synchronization since this algorithm has multiple reduction and serial steps.

ray tracing: A standard method to render photo-realistic 3D scenes is ray tracing. We implement a basic integer-based ray tracing engine in C and parallelize it with OpenMP. While each pixel can be rendered independently, ray tracing poses an interesting work-balancing challenge because it is a non-data-oblivious algorithm, meaning its execution trace depends on the input data. Specifically, the time to render each ray depends on the number of objects in its path. It is, therefore, a good example to show the benefit of individually programmable cores and the flexibility of MemPool.

Both algorithms run on a single-core and the full MemPool to demonstrate that MemPool can efficiently run complete applications with minimal parallelization overhead.

The *histogram equalization* application achieves 40% of the linear speedup and is mainly limited by its sequential parts and reductions, such as the computation of the transformation function. This speedup is very close to the ideal speedup considering Amdahl's law, illustrating how MemPool can parallelize complex algorithms with little overhead.

Ray tracing poses different challenges. It is fully parallelizable, but balancing the workload is challenging. Nevertheless, MemPool achieves 91% of the ideal speedup. 3% of the loss in performance comes from the imbalance in the workload distribution, while the remaining 6% can be found in the runtime overhead of OpenMP's dynamic scheduling. These applications show how MemPool enables implementing and running complex and data-dependent kernels with high-

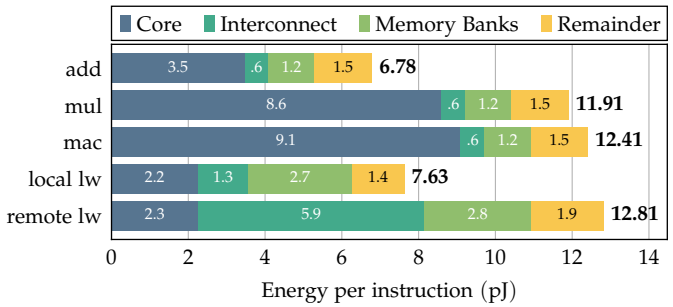


Fig. 16. Energy breakdown of individual instructions extracted from post-layout simulations. We measure the energy of three different arithmetic instructions and the load word instruction when loading from banks in the same (*local*) tile or *remote* ones, thereby passing through the groups. The *Remainder* category includes all components with very small contributions, mainly leakage, like the caches, DMA, AXI, etc.

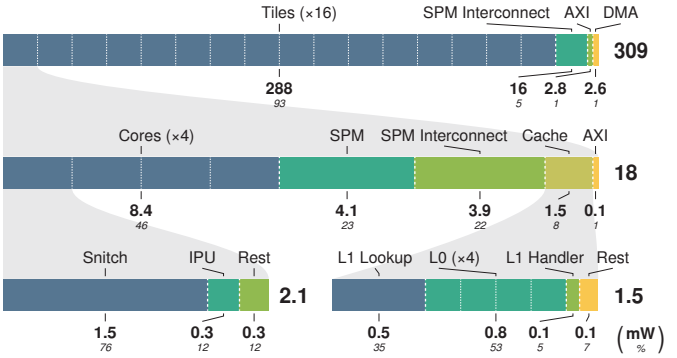


Fig. 17. Hierarchical power breakdown of MemPool when executing a *matmul* kernel. The results are shown in **mW** and % and are extracted using PrimeTime with annotations from a post-layout simulation.

level abstractions. Its shared memory and independent cores simplify parallelizing applications efficiently.

8.2.3 Energy of individual instructions

The energy consumed by different instructions quantifies the benefits of our ISA extensions and hybrid addressing scheme. Figure 16 shows the energy breakdown, including dynamic and static power, while all cores execute the instruction with randomized data. We report the energy per core per cycle since all instructions, including multi-cycle ones, are fully pipelined, and we execute one instruction per cycle per core.

The arithmetic instructions' energy consumption motivates our ISA supporting instructions like the MAC. Fusing the addition and multiplication increases a multiplications energy consumption by only 0.5 pJ, but eliminates the need for the add instruction, thereby saving 33%.

Similarly, the energy of local and remote memory transactions emphasizes the benefit of our hybrid addressing scheme. A remote transaction consumes 1.7× the energy of a local one. Therefore, the hybrid addressing scheme improves energy efficiency by reducing the number of remote accesses. Comparing remote loads to arithmetic instructions also shows how efficient MemPool's interconnect design is. Crossing the whole MemPool macro takes roughly the same energy as a MAC operation, thanks to the pipelined interconnect. Our interconnect remains energy efficient, despite being scaled to 256 initiators and 1024 targets.

8.2.4 Power breakdown

To gain deeper insights into MemPool's power consumption, Fig. 17 breaks the power consumption down into individual components. Despite scaling the cluster to 256 cores with low-latency access to L1, most of the power is still spent on computation. Specifically, 43% of the power is consumed by the cores, while the contribution of the SPM interconnect is kept at 21% despite its size, and the memory banks themselves account for another 21%.

9 Related Work

Energy-efficient manycore systems have been studied extensively, and various architectural approaches exist to tackle scaling parallel systems [1]. In the following, we focus on manycore systems based on clusters of PEs. Table 2 gives an overview of existing manycore systems.

A cluster of PEs sharing a low-latency L1 memory is a typical architecture for efficiently tackling today's parallel workloads. For example, Greenwaves' GAP9 processor's [11] compute power comes from nine independent RISC-V cores coupled to a shared L1 SPM. Its primary architectural idea is very similar to MemPool. However, MemPool presents a way to scale this idea to 256 cores. Ramon Chip's RC64 [13] targets task parallelism and implements a similar architecture to MemPool, coupling 64 DSP cores to a shared SPM accessible in a small number of cycles. Each core is equipped with a private instruction and data cache and a private SPM to reduce bandwidth requirements to the shared SPM. MemPool scales the single-cluster design even further without relying on costly data caches to reduce bandwidth requirements.

Other architectures rely on replicating the small compute cluster to scale to hundreds of cores. Manticore [14] is built out of a similar compute cluster as GAP9, coupling eight small 32-bit RISC-V cores to a low-latency shared L1 memory. This cluster is replicated and connected to a shared L2 memory via an AXI interconnect to build a system of 1024 PEs per chiplet. Kalray's MPPS3-80 [12] follows a similar approach with larger cores. 16 64-bit processors with private L1 data caches form a cluster with 4MB of L2 SPM memory. The chip features five clusters, resulting in 80 cores. Esperanto's ET-SoC-1 [15] has a similar hierarchy and has 1088 64-bit RISC-V cores organized in 34 *shires* of 32 cores and 4 MiB of SPM each. While this approach allows scaling into the manycore regime, it creates many small clusters with private memories, complicating the programming model and incurring communication overhead between clusters.

Modern GPUs resemble multi-cluster designs, with streaming multiprocessors (SMs) corresponding to shared-memory clusters that act as basic building blocks. NVIDIA's leading-edge H100 [4] features 144 SMs, each featuring 128 FP32 units. However, GPUs operate in the SIMT regime, meaning that a single instruction stream controls multiple compute units. Each SM has four scheduler and dispatch units controlling 32 FP32 units each, resulting in four independent instruction streams per SM or 576 per GPU. GPUs feature thousands of compute units, but their utilization is limited by the SIMT regime. Especially for irregular and non-data-oblivious algorithms, thread divergence and

TABLE 2
Comparison of MemPool with related, cluster-based architectures.

	Architecture	ISA	Cluster PEs	Total PEs	Independent PEs	Low latency interconnect	Open source
mesh-based	RAW [24]	32-bit MIPS-style	-	16	✓	✗	✗
	Celerity [25]	32-bit RISC-V	-	496*	✓	✗	✓
	KiloCore [26]	40-bit RISC	-	1000	✓	✗	✗
	Piton [27]	64-bit SPARC V9	-	25	✓	✗	✓
	TILE64 [28]	64-bit VLIW	-	64	✓	✗	✗
	Epiphany-V [29]	64-bit RISC	-	1024	✓	✗	✗
	Pixel Visual Core [6]	16-bit VLIW	256	2048	✗	✗	✗
crossbar-based	GAP9 [11]	32-bit RISC-V	9	9	✓	✓	≈
	RC64 [13]	32-bit VLIW	64	64	✓	✓	✗
	Manticore [14]	32-bit RISC-V	8	4096	✓	✗	✓
	MPPA3 [12]	64-bit VLIW	16	80	✓	✗	✗
	ET-SoC-1 [15]	64-bit RISC-V	32	1088	✓	✗	✗
	H1000 [4]	32/64-bit PTX	128	18432	✗	✗	✗
	This Work	32-bit RISC-V	256	256	✓	✓	✓

≈ = Closed source based on open source. *Contains additional five Linux-capable 64-bit cores and ten ultra-low-power cores.

synchronization restrict the GPU's performance. In contrast, MemPool gives each PE its instruction stream, making it much more flexible and efficient on irregular workloads.

An alternative way to scale to hundreds of PEs is to sacrifice the low-latency interconnect between memory banks and processors and directly connect neighboring cores through a 2D mesh (or similar, e.g., multi-mesh, Ruche, etc.) interconnect. Many multicore SoCs have been designed following this pattern [24], [25], [26], [27], [28], [29]. As an example, RAW [24] connects 16 small cores with private instruction and data memories in a 4×4 grid. The PEs are enhanced with neighbor-to-neighbor communication instructions, and sending data between the PEs furthest apart takes only six cycles. Celerity [25] and KiloCore [26] implement a similar architecture scaled to 496 or 1000 cores. Replication enables high core counts, but communication latency between distant cores quickly increases to 45 or 64 cycles, respectively, requiring careful dataflow management.

Piton [27] also connects cores in a mesh structure, but instead of using small cores, each of the 25 PEs consists of a Linux-capable 64-bit core with private instruction and data caches. TILE64 [28] and Epiphany-V [29] connect similar cores, caches, and DMAs in a 2D mesh but scale the architecture to up to 1024 cores. While all PEs still have access to other PE's memory, the network-on-chip (NoC) limits the inter-core bandwidth and imposes a high latency. Programming distributed cores requires distributing workloads in a spatially-aware fashion and carefully fitting local data within local memories to reach acceptable performance. Consequently, efficiently programming this class of architectures is challenging, especially for algorithms requiring PEs to access a shared pool of data. MemPool's low-latency interconnect greatly simplifies workload distribution and access to shared data, even when such accesses follow irregular and hard-to-predict patterns.

Similarly to KiloCore, Google's Pixel Visual Core [6] and TPU [30] connect their PEs in a 2D mesh to scale to 2048 or 64K PEs, respectively. The PEs are only equipped with private register files and communicate with their direct neighbors in a systolic fashion. While these architectures achieve very high performance for their specialized use cases, their rigid interconnect and programming model limit their flexibility.

In summary, MemPool is the first architecture to scale the shared-memory cluster with a low-latency interconnect to hundreds of individually programmable PEs. In contrast to smaller cluster-based architectures, MemPool allows for more parallelism and processing power while providing a larger L1 memory to reduce communication overhead and facilitate latency hiding from/to higher-level memories. We note that there is a physical limit to scaling a single cluster, and for extremely large core counts (thousands), moving to multiple clusters is inevitable. The MemPool architecture remains attractive even in a multi-cluster regime, as it reduces the need to partition shared data and greatly eases inter-cluster communication latency hiding with large block transfers.

10 Conclusion

MemPool presents a scalable, shared-L1-memory manycore RISC-V system. Even when scaling to 256 cores, all cores can access inter-tile or intra-cluster memory banks within at most five cycles of latency thanks to a sophisticated hierarchical SPM interconnect. Thanks to the capability to program each core independently, the presented cache hierarchy, and the refill interconnect, all cores can be highly utilized, i.e., up to 96% in common kernels for signal and image processing. Efficient data movement is implemented by a distributed DMA design integrated into the hierarchical design.

A full 256 core MemPool instance is implemented and evaluated in GlobalFoundries' 22FDX technology. It runs at 600 MHz (60 gate delays) in typical operating conditions (TT/0.80 V/25 °C) and occupies an area of 12.9 mm^2. By tailoring the Snitch cores to MemPool, and through enhancements like our DSP extension and hybrid addressing scheme, MemPool achieves very high performance and comes close to an ideally scaled system (e.g., full linear speedup).

Benchmarking on a comprehensive set of kernels shows that MemPool's performance is limited mainly by the inherent load-store architecture and synchronization associated with parallel programming. Architectural stalls contribute only a few percent of speedup loss. Thanks to a Halide framework and OpenMP runtime integration, the implementation of complex and irregular parallel kernels is straightforward, as illustrated with a ray tracing and histogram equalization implementation. In 22FDX technology a MemPool cluster achieves a performance up to 229 GOPS and an energy efficiency up to 192 GOPS/W. On average, access to shared memory only causes 4% stalls, and MemPool achieves a per-core IPC of 0.96 (over a theoretical bound of 1).

Acknowledgments

This work was supported by the ETH Future Computing Laboratory (EFCL), financed by a donation from Huawei Technologies. It also received funding from the HCSCA project #180625 funded by the Croatian-Swiss Research Programme, and by the European Union's Horizon 2020 research and innovation programme under grant agreement 101070374 (Convolve).

REFERENCES

[1] R. Muralidhar, R. Borovica-Gajic, and R. Buyya, "Energy efficient computing systems: Architectures, abstractions and modeling to techniques and standards," *ACM Comput. Surv.*, vol. 54, no. 11s, pp. 1–37, Sep. 2022.

[2] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing," *Brief. Bioinform.*, vol. 11, no. 5, pp. 473–483, May 2010.

[3] K. Rupp, "Microprocessor trend data," 2022. [Online]. Available: https://github.com/karlrupp/microprocessor-trend-data

[4] NVIDIA Corp., "NVIDIA H100 tensor core GPU architecture," NVIDIA Corp., Tech. Rep., 2022. [Online]. Available: https://www.nvidia.com/en-us/data-center/h100//

[5] K. Rocki *et al.*, "Fast stencil-code computation on a wafer-scale processor," in *Int. Conf. High Perform. Comput. Networking, Storage Anal.* Atlanta, Georgia: IEEE, Oct. 2020, p. 14.

[6] J. Redgrave, A. Meixner, N. Goulding-Hotta, A. Vasilyev, and O. Shacham, "Pixel Visual Core: Google's fully programmable image, vision and AI processor for mobile devices," in *2018 IEEE Hot Chips 30 Symp.*, Cupertino, US, Aug. 2018.

[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. San Francisco, CA, USA: Morgan Kaufmann, 2017.

[8] Apple Inc., "Apple unveils M1 Ultra, the world's most powerful chip for a personal computer," 2022. [Online]. Available: https://nr.apple.com/d2I1v3s8D5

[9] Intel Corporation, "Intel® core™ i9-12900ks processor," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/225916/intel-core-i912900ks-processor-30m-cache-up-to-5-50-ghz/specifications.html

[10] Ampere Computing Corp., "Ampere® Altra® 64-bit multi-core processor features," 2022. [Online]. Available: https://d1o0i0v5q5lp8h.cloudfront.net/ampere/live/assets/documents/Altra_Rev_A1_DS_v1.30_20220728.pdf

[11] GreenWaves Technologies SAS, "GAP9 next generation processor for hearables and smart sensors," GreenWaves Technologies SAS, Tech. Rep., 2021. [Online]. Available: https://greenwaves-technologies.com/wp-content/uploads/2022/06/Product-Brief-GAP9-Sensors-General-V1_14.pdf

[12] B. Dupont de Dinechin, "A qualitative approach to many-core architecture," in *Multi-Processor System-on-Chip 1: Architectures*, L. Andrade and F. Rousseau, Eds. Hoboken, New Jersey, USA: Wiley, Apr. 2021, ch. 2, pp. 27–51.

[13] R. Ginosar, P. Aviely, T. Israeli, and H. Meirov, "RC64: High performance rad-hard manycore," *IEEE Aerosp. Conf. Proc.*, pp. 2074–2082, Jun. 2016.

[14] F. Zaruba, F. Schuiki, and L. Benini, "Manticore: A 4096-core RISC-V chiplet architecture for ultra-efficient floating-point computing," *IEEE Micro*, vol. 41, no. 2, pp. 36–42, 2020.

[15] D. Ditzel *et al.*, "Accelerating ML recommendation with over a thousand RISC-V/tensor processors on Esperanto's ET-SoC-1 chip," in *2021 IEEE Hot Chips 33 Symp.* Palo Alto, California: IEEE, Aug. 2021, pp. 209–220.

[16] M. Cavalcante, S. Riedel, A. Pullini, and L. Benini, "MemPool: A shared-L1 memory many-core cluster with a low-latency interconnect," in *2021 Des. Autom. Test Eur. Conf. Exhib.* Grenoble, France: IEEE, 2020, pp. 701–706.

[17] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Decoupling algorithms from schedules for easy optimization of image processing pipelines," *ACM Trans. Graph.*, vol. 31, no. 4, pp. 1–12, Jul. 2012.

[18] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads," *IEEE Trans. Comput.*, vol. 70, no. 11, pp. 1845–1860, Feb. 2021.

[19] S. Mazzola, S. Riedel, M. Cavalcante, L. Benini, and A. Macii, "ISA extensions in the Snitch processor for signal processing," Master's thesis, Politecnico di Torino, Apr. 2021.

[20] W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann, 2004.

[21] A. Kurth *et al.*, "An open-source platform for high-performance non-coherent on-chip communication," *IEEE Trans. Comput.*, vol. 71, no. 8, pp. 1794–1809, Sep. 2022.

[22] S. Riedel, F. Schuiki, P. Scheffler, F. Zaruba, and L. Benini, "Banshee: A fast LLVM-based RISC-V binary translator," in *IEEE/ACM Int. Conf. Comput. Des.* IEEE, Nov. 2021, pp. 1105–1113.

[23] G. Tagliavini, D. Cesarini, and A. Marongiu, "Unleashing fine-grained parallelism on embedded many-core accelerators with lightweight OpenMP tasking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 9, pp. 2150–2163, Sep. 2018.

[24] M. B. Taylor *et al.*, "The Raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, Mar. 2002.

[25] S. Davidson *et al.*, "The Celerity open-source 511-core RISC-V tiered accelerator fabric: Fast architectures and design methodologies for fast chips," *IEEE Micro*, vol. 38, no. 2, pp. 30–41, Mar. 2018.

[26] B. Bohnenstiehl *et al.*, "KiloCore: A 32-nm 1000-processor computational array," *IEEE J. Solid-State Circuits*, vol. 52, no. 4, pp. 891–902, Apr. 2017.

[27] M. McKeown *et al.*, "Piton: A manycore processor for multitenant clouds," *IEEE Micro*, vol. 37, no. 2, pp. 70–80, Mar. 2017.

[28] S. Bell *et al.*, "TILE64™ - processor: A 64-core SoC with mesh interconnect," in *Dig. Tech. Pap. - IEEE Int. Solid-State Circuits Conf.*, vol. 51. San Francisco, CA, USA: IEEE, 2008, pp. 87–89.

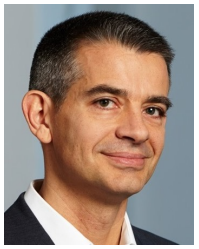
[29] A. Olofsson, T. Nordström, and Z. Ul-Abdin, "Kickstarting high-performance energy-efficient manycore architectures with Epiphany," in *Conf. Rec. - Asilomar Conf. Signals, Syst. Comput.* IEEE, Apr. 2015, pp. 1719–1726.

[30] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. - Int. Symp. Comput. Archit.* Toronto, ON, Canada: IEEE, 2017, pp. 1–12.

Samuel Riedel received the B.Sc. and M.Sc. degree in Electrical Engineering and Information Technology at ETH Zurich in 2017 and 2019, respectively. He is currently pursuing a Ph.D. degree with the Digital Circuits and Systems group of Luca Benini. His research interests include computer architecture, focusing on manycore systems and their programming model.

Matheus Cavalcante received the M.Sc. degree in Integrated Electronic Systems from the Grenoble Institute of Technology (Phelma) in 2018, and is since then pursuing the Ph.D. degree at ETH Zurich, with the Digital Circuits and Systems Group of Prof. Luca Benini. His current research interests include the design of very-large-scale circuits and high-performance systems, namely vector and manycore architectures, and their co-optimization with emerging VLSI technologies.

Renzo Andri received the B.Sc., M.Sc., and Ph.D. degree in Electrical Engineering and Information Technology at ETH Zurich in 2013, 2015, and 2020, respectively. His research focuses on energy-efficient machine learning acceleration from system-level design to full-custom IC design. In 2019, he won the IEEE TCAD Donald O. Pederson Award.



Luca Benini is the Chair of Digital Circuits and Systems at ETH Zürich and a Full Professor at the University of Bologna. He has served as Chief Architect for the Platform2012 in STMicroelectronics, Grenoble. Dr. Benini's research interests are in energy-efficient systems and multi-core SoC design. He is also active in the area of energy-efficient smart sensors and sensor networks. He has published more than 1000 papers, five books and several book chapters. He is a Fellow of the ACM and a member of the Academia Europaea.