ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

MemPool: A Shared-L1 Memory Many-Core Cluster with a Low-Latency Interconnect

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:
Cavalcante, M., Riedel, S., Pullini, A., Benini, L. (2021). MemPool: A Shared-L1 Memory Many-Core Cluster with a Low-Latency Interconnect. Institute of Electrical and Electronics Engineers Inc. [10.23919/DATE51398.2021.9474087].

Availability:
This version is available at: https://hdl.handle.net/11585/870440 since: 2022-02-26

Published:
DOI: http://doi.org/10.23919/DATE51398.2021.9474087

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

17 October 2026

MemPool: A Shared-L1 Memory Many-Core Cluster with a Low-Latency Interconnect

Matheus Cavalcante
ETH Zürich
Zürich, Switzerland
matheusd *at* iis.ee.ethz.ch

Samuel Riedel
ETH Zürich
Zürich, Switzerland
sriedel *at* iis.ee.ethz.ch

Antonio Pullini
GreenWaves Technologies
Grenoble, France
pullinia *at* iis.ee.ethz.ch

Luca Benini
ETH Zürich
Zürich, Switzerland
Università di Bologna
Bologna, Italy
lbenini *at* iis.ee.ethz.ch

***Abstract*—A key challenge in scaling shared-L1 multi-core clusters towards many-core (more than 16 cores) configurations is to ensure low-latency and efficient access to the L1 memory. In this work we demonstrate that it is possible to scale up the shared-L1 architecture: We present MemPool, a 32 bit many-core system with 256 fast RV32IMA "Snitch" cores featuring application-tunable execution units, running at 700 MHz in typical conditions (TT/0.80 V/25 °C). MemPool is easy to program, with all the cores sharing a global view of a large L1 scratchpad memory pool, accessible within at most 5 cycles. In MemPool's physical-aware design, we emphasized the exploration, design, and optimization of the low-latency processor-to-L1-memory interconnect. We compare three candidate topologies, analyzing them in terms of latency, throughput, and back-end feasibility. The chosen topology keeps the average latency at fewer than 6 cycles, even for a heavy injected load of 0.33 request/core/cycle. We also propose a lightweight addressing scheme that maps each core private data to a memory bank accessible within one cycle, which leads to performance gains of up to 20 % in real-world signal processing benchmarks. The addressing scheme is also highly efficient in terms of energy consumption since requests to local banks consume only half of the energy required to access remote banks. Our design achieves competitive performance with respect to an ideal, non-implementable full-crossbar baseline.**

***Index Terms*—Many-core; MIMD; Networks-on-Chips.**

I. Introduction

The failure of Dennard scaling [1] has implied a power wall for computing, limiting processor frequencies [2]. To achieve high performance under a limited power budget, core count scaling has been used instead. Multi-core architectures are the norm today, allowing for high performance and efficient computing on a wide range of applications.

There are many flavors of multi-core architectures. Some consist of a few general-purpose high-performance cores sharing a large cache, such as Arm's Cortex-A77 [3] and Intel's Core-i9 processors [4]. Others are highly specialized processor arrays, usually with a specialized interconnection network adapted to the intended application domain [5], such as Google's Pixel Visual Core [6], an Image Processing Unit (IPU) with a 2D mesh network of 256 computing units that communicate with near-neighbor fixed-size messages.

We focus on a common architectural pattern for building multi-core architectures, namely a cluster of simple cores sharing L1 memory through a low-latency interconnect [7]. We can find instances of this architectural pattern across many different domains, from the streaming processors of Graphics Processing Units (GPUs) [8], to the ultra-low-power domain with GreenWaves' GAP8 processor [9], to high-performance embedded signal processing with the Kalray processor clusters [10], to aerospace applications with the Ramon Chips' RC64 system [11]. However, as we will detail in Section II, these clusters only scale to low tens of cores and suffer from long memory access latency due to the topology of their interconnects.

In this paper, we set out to design and optimize the first scaled-up *many-core* system with shared low-latency L1 memory. To this end, we propose MemPool, a 32 bit RISC-V-based system, with 256 small cores sharing a large pool of Scratchpad Memory (SPM). In the absence of contention, all the SPM banks are accessible within 5 cycles. The contributions of this paper are:

- The physical-aware design of MemPool's architecture, with particular emphasis on the exploration, design, and optimization of a low-latency processor-to-L1-memory interconnection network (Section III);
- The creation of a lightweight and transparent memory addressing scheme that keeps the memory region most often accessed by a core—e.g., the stack—in a memory bank close by, with minimal access latency (Section IV);
- The complete physical implementation and performance, power, and area analysis of a large cluster in an advanced GlobalFoundries 22FDX Fully Depleted Silicon on Insulator (FD-SOI) technology node (Sections V and VI).

MemPool runs at 700 MHz in typical conditions (480 MHz in worst-case conditions). The critical path of the design is dominated by wire delay (37 %), with 27 out of its 36 gates being either buffers or inverter pairs. Its processor-to-L1 interconnect has an average latency of fewer than 6 cycles, even for a heavy injected load of 0.33 request/core/cycle. Our addressing scheme helps to keep the memory requests in local banks accessible within one cycle, which leads to performance gains up to 20 % in real-world benchmarks. This scheme is also highly effective in terms of energy consumption since local memory requests consume only half of the energy required for remote memory accesses. In a nutshell, we demonstrate in this paper that we can scale the core count of an L1-shared cluster to ten times more cores than what was previously considered achievable, with cycle counts on various benchmarks that are comparable with an ideal, non-implementable full-crossbar baseline.

II. Related Work

In this work, we focus on architectures that feature a cluster of simple cores sharing low-latency L1 memory [7], which is a very common architectural pattern. The latest Nvidia Ampere GPUs, for example, have "streaming multiprocessors" with 32 double-precision floating-point cores each, sharing 192 KiB of L1 memory [12]. Similar architectural patterns can also be found in the embedded and ultra-low-power domain. GreenWaves' GAP8 [9] is an Internet-of-Things (IoT) application-class processor with eight cores sharing 128 KiB of L1 memory. The Snitch cluster [13] features a cluster of eight RV32IMA cores sharing 128 KiB of private SPM accessible within 2 cycles.

It is commonly believed that the number of cores in a single L1-shared cluster is bound to the low-tens limit. To scale the core count of many-core systems into the hundreds, memory sharing is usually done at some high-latency hierarchy level. Moving to multiple clusters creates challenges in terms of programmability [10]: tile-based systems are usually connected by meshes that have long access latency and usually require non-uniform memory access (NUMA) models of computation.

An example of multi-cluster design is Kalray's MPPA-256. It integrates 256 user cores into 16 clusters. Each MPPA-256 cluster has its own private address space [10], with the clusters connected by specialized Network-on-Chips (NoCs). Memory sharing is done at the level of main memory. This design achieves high efficiency by compromising on memory sharing between clusters, thus circumventing the need for low-level cache coherence protocols. Tilera's TILE-Gx series, on the other hand, equips each tile processor with an L2 cache, and inter-tile and main memory communication are provided by five 2D mesh networks [14]. Ramon Chips' RC64 system brings this design pattern to the harsh aerospace conditions [11]. Its 64 cores have private SPMs, and share access to 4 MiB of on-chip memory, accessible through a logarithmic network.

Clearly, there is a push toward high core-count many-core architectures to tackle embarrassingly parallel problems, such as image processing and machine learning. Google's Pixel Visual Core [6], for example, is an IPU with specialized stencil processors interconnected with a ring network. Within each stencil processor, an array of 256 lanes communicate through a rigid read-neighbor network. This design is highly specialized for systolic algorithms, which can take advantage of data sharing between neighbors. The rigidity of memory allocation and the long access latency reduce the applicability of this highly efficient design to other non-systolic algorithms.

The main novelty of our work is to demonstrate we can scale up the shared-L1 processor cluster in a region that was considered to be completely infeasible by all previous related works, namely hundreds of cores, with extremely competitive performance and efficiency.

III. Architecture

MemPool has a large multi-banked pool of L1 memory, shared among all the cores. A fully connected non-blocking logarithmic interconnect between the 256 cores would be infeasible due to routing congestion and area scaling [15]. In this work, we show this can be overcome through a hierarchical approach at the topological and physical level, placing the cores and memory banks in a regular array connected by low-diameter networks. The following sections go bottom-up over the main hierarchical modules that compose MemPool.

A. Interconnect architecture

MemPool has two parallel interconnects, one routing the cores' requests to the SPM banks, and one routing read responses back. The basic element of both interconnects is a single-stage $m \times n$ crossbar switch, connecting m masters to n slaves. An optional elastic buffer can be inserted at each output of the switch, after address decoding and round-robin arbitration, to break any combinational paths crossing the switch [16].

With the access latency constraint in mind, the interconnect building blocks are kept as streamlined as possible. They do not provide transaction ordering, with this task offloaded to the cores. Moreover, since they are used to transmit single-word-requests from the cores, the network does not use virtual channels. We use oblivious routing since there is a single path for each master/slave pair. In terms of network topology, two-dimensional mesh networks were discarded due to their bad latency scaling. We also avoided Ogras and Marculescu's approach [17], which optimizes the latency of a mesh network by iteratively adding long-ranging links to the topology, but requires an advanced routing algorithm. MemPool uses a combination of fully-connected crossbars and minimal radix-4 butterfly networks, whose topology can be seen in Figure 1.

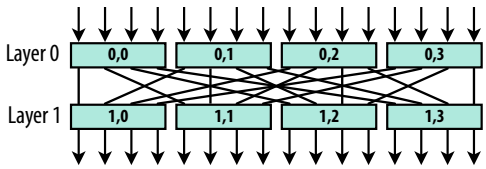


Fig. 1. Topology of a 16×16 radix-4 butterfly. The boxes represent the base routing element, a 4×4 logarithmic crossbar switch.

B. Tile

At the base of MemPool's hierarchy, we have tiles, whose architecture can be seen in Figure 2. The tiles are based on Snitch, a 21 kGE single-issue single-stage RISC-V-based RV32IMA core [13], whose small area allows for massive replication. Snitch supports a configurable number of outstanding load instructions, which is useful to hide the SPM access latency.

The tile contains 16 memory banks, and each core has a dedicated port to access them with one cycle latency. The cores share K master ports to access remote tiles. An address decoder at the output of the cores statically decides where to send the cores' requests. Each tile has K slave request ports, receiving memory requests from remote tiles. There is a register boundary at the master request and response ports. Both request and response interconnects are realized as fully-connected crossbars. Requests hold metadata to route them back to the correct core and ensure their proper ordering by the Reorder Buffer (ROB).

Inside each tile, we have a 4-way L1 instruction cache. The cache has a 32 bit Advanced eXtensible Interface (AXI) refill

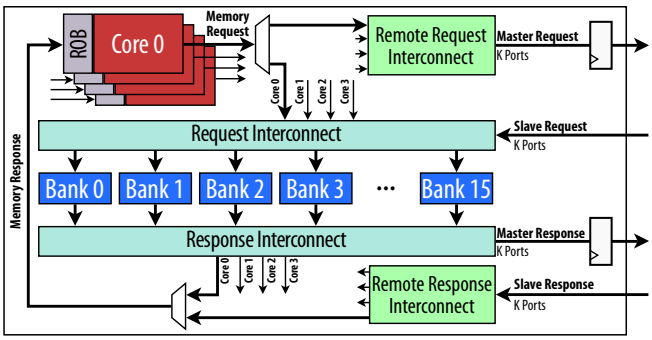


Fig. 2. Architecture of MemPool tiles, with K request ports and K response ports to remote tiles.

port. These ports can be connected to a low-overhead refill-network (e.g., a ring), which is noncritical, and hence it is not further discussed in this work.

C. MemPool cluster

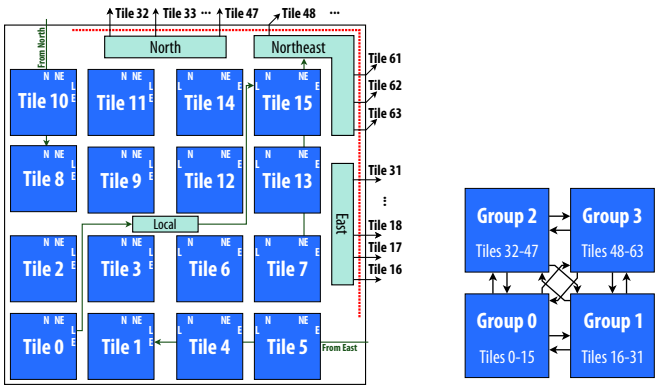
We evaluated three network topologies for the global interconnection network between tiles.

1) Top_1 – single 64×64 radix-4 butterfly: In this configuration, each tile has a single remote port for communication with remote tiles, i.e., $K = 1$. A single 64×64 radix-4 butterfly network, with a single pipeline stage midway through its $\log_4(64) = 3$ layers, connects the tiles. Therefore, data in any remote memory bank can be accessed within 5 cycles. Inside the tile, two 4×1 crossbars arbitrate the core requests and the memory responses. This design creates a bottleneck at the tile boundary, since the traffic of 4 cores is concentrated through a single port.

2) Top_4 – four parallel 64×64 radix-4 butterflies: To reduce the traffic bottleneck at the tile boundary, we evaluated a system that replicates the global 64×64 butterfly interconnect four times. That is, each tile has four master request and response ports, each associated with their own 64×64 interconnect. Each master request port is dedicated to a core, i.e., the remote request interconnect is effectively a point-to-point connection.

3) Top_H – hierarchical approach: Both Top_1 and Top_4 have a uniform access pattern, with a 5 cycle access latency between any two tiles. Hence, requests between two tiles need to cross the whole interconnect, regardless of how physically close they are. This leads to decreased efficiency and increased routing congestion due to longer paths towards the center of the design. We introduce a new level of hierarchy to maintain the bandwidth of Top_4, while avoiding long detours between neighboring tiles. Figure 3a shows a *local group* of 16 tiles. Cores access remote memory banks in the same local group within 3 cycles thanks to the local interconnect, a 16×16 fully-connected crossbar.

The MemPool cluster is composed of four local groups, as shown in Figure 3b. Inside each local group, the *north* (N), *northeast* (NE), and *east* (E) 16×16 radix-4 butterfly networks are responsible for communication between local groups. Each tile has corresponding N, NE, and E ports, and a *local* (L) port to access tiles within the same local group. A 4×4 crossbar inside each tile routes the requests to the correct port. There



(a) Local group. (b) Cluster.

Fig. 3. MemPool's Top_H architecture. Dashed lines indicate a register boundary.

is a register boundary at the local groups' master interfaces, increasing the zero-load access latency of a memory bank in a remote local group to 5 cycles.

IV. Hybrid Addressing Scheme

MemPool has a sequentially interleaved memory mapping across all memory banks in order to minimize banking conflicts. However, this also implies that most memory requests target remote tiles. Optimally, all cores' requests would remain in the local tile, which would lower the latency and power consumption. With the scrambling logic, visualized in Figure 4, we transform an interleaved memory map into a *hybrid* one, by adding *sequential regions* in which contiguous addresses target a single tile. The top half shows the classical fully interleaved memory scheme. The address and memory map at the bottom is a hybrid memory map created by swapping the address bits.

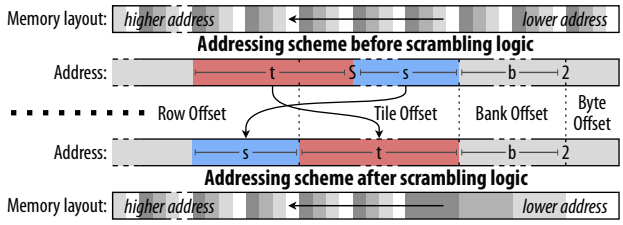


Fig. 4. Hybrid addressing scheme via the scrambling logic. The upper and lower parts show the fully interleaved and the hybrid memory map, respectively. The outer bars visualize the memory map, with the shades representing different tiles to which the addresses are mapped. The inner bars are the addresses, with the scrambling in between leading from one scheme to the other.

With 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 the 2^b banks of each tile. The next t bits distinguish between the 2^t tiles. The remaining address bits are interpreted as the row offset within each bank.

Consider each tile with a sequential memory region of 2^S bytes, or 2^s rows in the tile's banks. Since the banks inside the same tile are still accessed interleaved, we leave the byte and bank offsets untouched. The next s bits represent part of the tile offset, but we need them to represent the banks' next row within the same tile. Therefore, we shift them t bits to the left—where

the row offset starts—and fill them with the t bits we replaced. This creates 2^t sequential regions, one for each tile. In total, we dedicate the first 2^{S+t} bytes to sequential regions. We leave the subsequent bytes interleaved by conditionally applying the scrambling to addresses inside the sequential memory region.

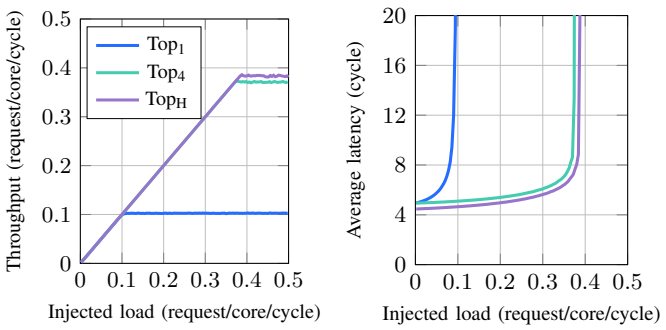
The hybrid addressing scheme's key benefit is giving the programmer the additional capability to store private data, such as a core's stack, in the same tile. It reduces the number of transactions between tiles,making better use of the tiles' fully-connected, high-throughput crossbar. Sequential memory regions are prone to banking conflicts. However, by only mapping private data to the sequential region, the cores' accesses remain distributed across all banks. In contrast to aliasing or completely private memories, we do not complicate programmability but, by applying the same address transformation for all cores, give all cores the same memory view and keep the L1 memory region contiguous and shared. The beneficiary of the sequential region are programs that make heavy use of the stack or work mainly on local data. The scrambling logic can be efficiently implemented in hardware with a wire crossing and a multiplexer.

V. Performance analysis

A. Interconnect architecture

In this section, we analyze the three proposed network topologies in terms of average latency and throughput, as a function of the injected load λ (measured in requests per core per cycle). The results were extracted using an extensive cycle-accurate Register Transfer Level (RTL) simulation. Each core is replaced by a synthetic traffic generator, which generates new requests following a Poisson process of rate λ. The requests have a random uniformly distributed destination memory bank.

Figure 5a shows the different topologies' throughput, with an increasing load. At a 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.



(a) Throughput. (b) Average latency.

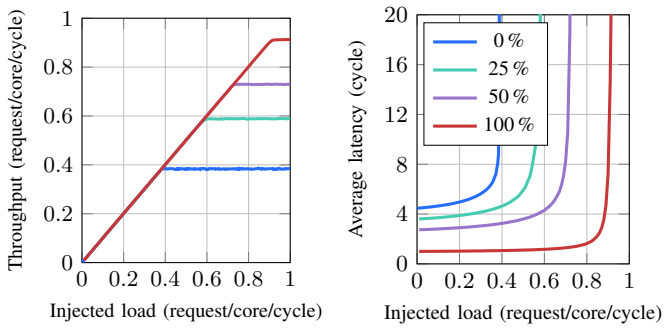
Fig. 5. Network analysis of the three proposed network topologies, in terms of throughput and average round-trip latency, as a function of the injected load.

As a counterpart to Figure 5a, Figure 5b shows the average round-trip latency of the requests for an increasing load. It elevates the point where the topologies become congested by showing the explosion of the average latency. The average latency of Top_H 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. Both results imply that the Top_1's traffic concentration at the tiles' ports leads to unacceptable performance degradation.

B. Hybrid addressing scheme

To evaluate the performance impact 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 $1 - p_{\text{local}}$.

Figure 6a 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, or using local memory in general, can vastly improve the system's throughput by preventing the congestion in the global interconnect, besides lowering the overall average access latency as seen in Figure 6b. An application making 25 % of its accesses to the stack, mapped at the sequential region, can gain up to 50 % in performance by using the scrambling logic, without changing the code.



(a) Throughput. (b) Average latency.

Fig. 6. Network analysis of Top_H with our hybrid addressing scheme, as a function of the injected load, for different probabilities of requesting data in the local tile's sequential region p_{local}.

C. Benchmarks

In this section, we benchmark MemPool with three real-world highly-parallelizable signal processing benchmarks' runtime:

matmul: a matrix multiplication of two 64×64 matrices, for which accesses are predominantly remote;

2dconv: a 2D discrete convolution with a 3×3 kernel, for which all accesses are local, except for cores working on windows that require data from two tiles;

dct: a 2D discrete cosine transform (DCT) operating on 8×8 blocks residing in local memory. It uses the stack to store intermediate results, i.e., all accesses are local, given the stack is mapped to local banks.

We derive our baseline systems from a Snitch cluster [13], which we ideally scale up to a cluster of 256 cores connected to 1024 banks through a fully-connected crossbar. The systems assume an idealized interconnect with no routing conflicts that allows all banks to be accessed within one cycle. This idealized network is physically infeasible due to high routing congestion with reasonable clock rates. We use two baseline systems, with

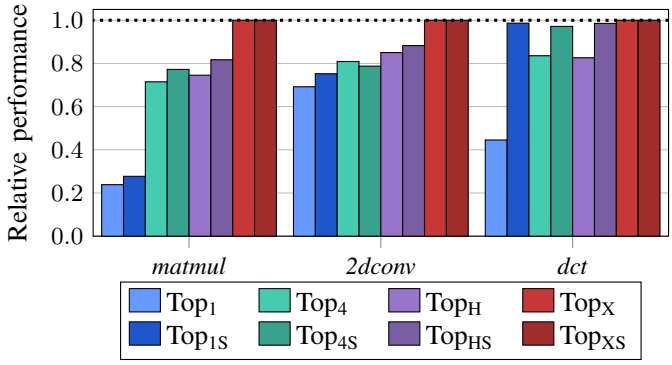


Fig. 7. Performance of the three benchmarks on all topologies, relative to the baselines.$\text{Top}_{\blacklozenge\text{S}}$ represents the $\text{Top}_{\blacklozenge}$ with scrambling logic.

(Top_{XS}) and without (Top_{X}) scrambling logic, to compare to the respective $\text{Top}_{\blacklozenge\text{S}}$ and $\text{Top}_{\blacklozenge}$ MemPool systems (where $\text{Top}_{\blacklozenge}$ stands for any of the topologies defined in Section III-C).

Figure 7 shows the benchmarks' performance, normalized by the performance achieved on the baseline architectures. Top_{H} generally beats Top_4, and they both outperform Top_1 by a factor of three in the extreme cases. A big performance difference is visible in *matmul*, which has many remote accesses. For *dct*, the three topologies with the scrambling logic perform equally well, as they all operate on data mapped to local banks. Without the scrambling logic, the stacks become spread over all tiles, leading to a significant performance penalty, especially for Top_1.

Top_{H} performs very close to the baseline for all benchmarks, with a performance penalty of at most 20 % for *matmul* due to its non-local access pattern. The hybrid memory map increases the locality of the data accesses, improving overall performance. With *dct*, we match the baseline since we only do local accesses.

VI. Physical Implementation

In this section, we analyze the feasibility of MemPool using the Top_1, Top_4, and Top_{H} topologies. We also analyze them in terms of power, performance, and area results.

A. Methodology

MemPool was synthesized for GlobalFoundries 22FDX FD-SOI technology using Synopsys Design Compiler Graphical 2019.12. We used floorplanning information during synthesis to improve timing correlation with the back-end design. Each tile has 2 KiB of instruction cache, and 16 KiB of SPM—i.e., the MemPool cluster has 1 MiB of L1 SPM. The back-end flow was carried out with Synopsys IC Compiler II 2019.12, targeting 500 MHz at worst-case conditions (SS/0.72 V/125 °C). MemPool's power results were extracted with switching activities obtained by simulating the benchmarks on a netlist back-annotated with post-place-and-route delay information. We used Synopsys PrimeTime 2019.12 to carry out the sign-off timing extraction at worst-case conditions and power analysis at typical conditions (TT/0.80 V/25 °C).

B. Tile implementation

Due to the size and the regularity of MemPool's design, we used a hierarchical implementation flow. The tile was implemented as a square 425 µm × 425 µm macro (908 kGE). The most complex tile—i.e., Top_{H}'s tile—is shown in Figure 8.

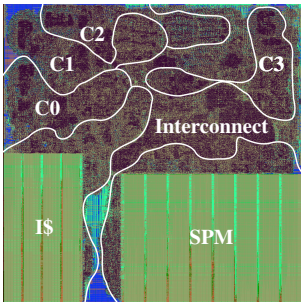


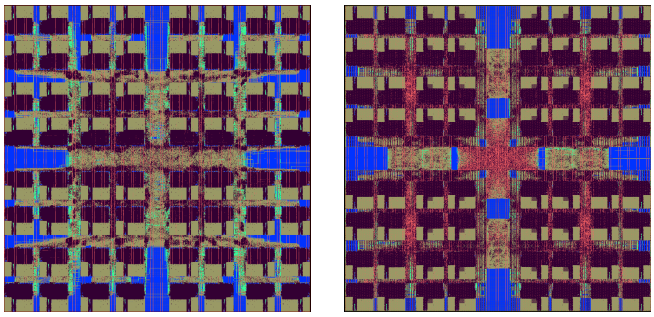
Fig. 8. Placed and routed Top_{H} tile, as a 425 µm × 425 µm macro.

The tile's critical path (53 gates long) starts at a register after the instruction cache, passing through the 2nd Snitch core and the request interconnect, before arriving at a SPM bank. The tile achieves a utilization of 72.8 %, the area is dominated by the instruction cache (23.6 %) and by the L1 SPM (40.2 %).

C. MemPool cluster implementation

The MemPool cluster is implemented as a 4.6 mm × 4.6 mm macro, i.e., 55 % of the design area is covered by the tiles. The area overhead was driven by congestion, which is the main constraint of the design, particularly at the center of the design.

Figure 9a shows the placed-and-routed Top_1 macro. With its 64 × 64 radix-4 butterfly topology, the connection between any two remote tiles needs to cross the whole network, regardless of the physical distance between the tiles. Therefore, all wiring and cells are drawn towards the center of the design, which is heavily congested. Top_4 is four times more congested than Top_1, which is enough to make it physically infeasible with reasonable clock rates. The placed-and-routed Top_{H} macro can be see in Figure 9b. Similarly to Top_1, there is a high cell and wiring density at the center of the design, due to the connection between the two diagonally placed groups (Figure 3b). However, unlike $\text{Top}_{1/4}$, Top_{H} distributes the cells and the wiring throughout the cluster, through the use of the directional local group interconnects.



(a) Top_1. (b) Top_{H}.

Fig. 9. Placed and routed Top_1 and Top_{H} MemPool clusters, implemented as 4.6 mm × 4.6 mm macros. The dark blue regions are devoid of standard cells.

Top_4 and Top_{H} achieve much better performance results (in terms of latency and throughput) than Top_1. However, out of these two high-performance topologies, only Top_{H} is physically

feasible. The Top_H MemPool cluster runs at 700 MHz at typical conditions (480 MHz at worst-case conditions). The critical path of this design starts at the boundary of one local group, passes through the center of the cluster and another local group until reaching the ROB of a Snitch core. Wire propagation delay accounts for 37 % of the timing of the critical path, and 27 out of the 36 gates in this path are either buffers or inverter pairs.

D. Power analysis

In this section, we analyze the power and energy consumption of the Top_H MemPool cluster, while running the *matmul* kernel at 500 MHz, in typical operating conditions (TT/0.80 V/25 °C). Each tile consumes, on average, 20.9 mW. The main culprits are the instruction cache, at 8.3 mW (39.5 % of the tile's total power consumption), the Snitch cores, at 5.6 mW (26.6 %), and the SPM banks, at 2.6 mW (12.6 %). The request and response interconnects only consume 1.7 mW, less than 10 % of the tile's total power consumption. At the top level, MemPool consumes 1.55 W, 86 % of which being consumed within the tiles.

Figure 10 summarizes the energy consumption per instruction of the Top_H tile, for different instructions. Each local load uses 8.4 pJ. About half of this energy consumption, 4.5 pJ, is spent at the local interconnect. Remote loads use the global interconnect, which raises their energy consumption to 16.9 pJ. In this case, the interconnects consume 13.0 pJ, or 2.9× the energy consumed at the interconnects for a local load.

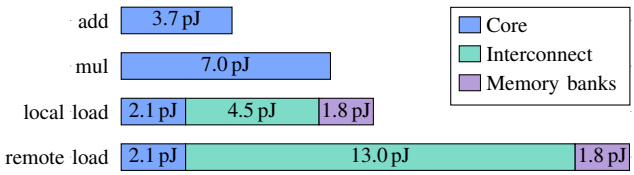


Fig. 10. Breakdown of Top_H's energy consumption per instruction.

As a comparison with arithmetic instructions, a local load uses about as much energy as a complex instruction such as mul, or 2.3× the energy consumed by a simple add. Remote loads have the highest energy requirements, but even then that is only 4.5× the energy of an add. This result confirms that MemPool is a balanced design that is not severely interconnect-dominated.

VII. Conclusion

In this paper, we presented MemPool, a 32 bit system with 256 ultra-small RV32IMA Snitch cores sharing 1 MiB of L1 SPM. MemPool was implemented in GlobalFoundries 22FDX FD-SOI technology, achieving 700 MHz in typical conditions. MemPool's architecture was driven by a physical-aware analysis of three different low-latency processor-to-L1-memory interconnect topologies. We chose the one that leads to the best performance results in terms of throughput and average latency, while also being physically feasible. In the absence of contention, all SPM banks are accessible within 5 cycles.

We compared MemPool's performance with a baseline system that has an idealized crossbar switch. Our system achieves at least 80 % of the baseline's performance on real-world signal processing benchmarks. Our hybrid addressing scheme helps keep the memory requests in local banks accessible in one cycle, leading to performance gains up to 20 % in such benchmarks.

Similarly to tile-based systems (which use 2D mesh NoCs), this scheme provides low-latency access to a memory range. However, our scheme has two advantages: (a) no aliasing, so that we can use this local range with more flexibility, and (b) much lower latency and higher bandwidth for all the global accesses, which enables us to run "non-systolic" algorithms effectively. The addressing scheme is also highly efficient in terms of energy consumption since local memory requests consume only half of the energy required for remote accesses.

References

[1] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct. 1974.

[2] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, Sep. 2011, pp. 365–376.

[3] Arm Corp., *Arm Cortex-A77 Core*, Cambridge, UK, Oct. 2019, revision r1p1.

[4] Intel Corp., "Intel Core i9-10900X X-Series processor," Oct. 2019. [Online]. Available: https://www.intel.com/content/www/us/en/products/processors/core/x-series/i9-10900x.html

[5] G. Blake, R. G. Dreslinski, and T. Mudge, "A survey of multicore processors," *IEEE Signal Processing Magazine*, vol. 26, no. 6, 2009.

[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 Symposium (HC30)*. Cupertino, US: IEEE Technical Committee on Microprocessors and Microcomputers, Aug. 2018.

[7] H. Ayed, J. Ermont, J.-L. Scharbarg, and C. Fraboul, "Towards a unified approach for worst-case analysis of Tilera-like and KalRay-like NoC architectures," in *2016 IEEE World Conference on Factory Communication Systems (WFCS)*, May 2016.

[8] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar. 2008.

[9] GreenWaves Technologies Corp., *GAP8 Hardware Reference Manual*, Grenoble, FR, Jan. 2019, version 1.5.5. [Online]. Available: https://gwt-website-files.s3.amazonaws.com/gap8_datasheet.pdf

[10] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel, "A distributed run-time environment for the Kalray MPPA-256 integrated manycore processor," *Procedia Computer Science*, vol. 18, pp. 1654 – 1663, 2013, 2013 International Conference on Computational Science.

[11] R. Ginosar, P. Aviely, T. Israeli, and H. Meirov, "RC64: High performance rad-hard manycore," in *2016 IEEE Aerospace Conference*, Mar. 2016.

[12] Nvidia Corp., *Nvidia A100 Tensor Core GPU Architecture*, 1st ed., 2020. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf

[13] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini. (2020, Feb.) Snitch: A 10 kGE pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads. arXiv:2002.10143v1 [cs.AR].

[14] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. Miao, J. F. Brown III, and A. Agarwal, "On-chip interconnection architecture of the Tile processor," *IEEE Micro*, vol. 27, no. 5, pp. 15–31, Nov. 2007.

[15] G. Michelogiannakis, J. Balfour, and W. J. Dally, "Elastic-buffer flow control for on-chip networks," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, Feb. 2009, pp. 151–162.

[16] G. Dimitrakopoulos, A. Psarras, and I. Seitanidis, *Microarchitecture of Network-on-Chip Routers: A Designer's Perspective*. Springer Publishing Company, 2014.

[17] U. Y. Ogras and R. Marculescu, "'It's a small world after all': NoC performance optimization via long-range link insertion," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 7, pp. 693–706, Jul. 2006.