PyCacheGen: A Highly Configurable Open-Source Generator for Synthesizable Caches

Richard Müller[1,†], Konstantin Lübeck[1,*,†], Michael Kuhn[1], Paul Palomero Bernardo[1] and Oliver Bringmann[1]

[1]*University of Tübingen, Embedded Systems, Sand 13, 72076 Tübingen, Germany*

Abstract

Caches are essential components of modern computer architectures, playing a crucial role in bridging the performance gap between fast processors and relatively slow memory. However, designing and integrating highly configurable caches into complex system-on-chip (SoC) designs remains a significant challenge. To address this, we present PyCacheGen, a highly configurable open-source generator for synthesizable caches.

PyCacheGen enables the generation of synthesizable Verilog cache modules with a broad range of configurable parameters, including associativity, write policy, allocation policy, write buffer, and multiple request ports. Notably, the caches generated by PyCacheGen support the write-back policy, which can lead to substantial performance improvements over the write-through policy. To showcase the capabilities of PyCacheGen, we generated various cache designs and successfully integrated them into the RISC-V PULPissimo SoC and synthesized the designs using GlobalFoundries' 22FDX+ technology node.

Our results demonstrate that caches generated by PyCacheGen can significantly enhance performance for slow memory hierarchies and reduce energy by up to 5.51% for unit latency memories, all while increasing the area of the PULPissimo SoC by only 0.57%.

1. Introduction

As modern processor speeds have increased dramatically over the years, the latency for accessing data from memories has not kept pace, leading to a bottleneck in overall system performance known as the processor-memory performance gap or memory wall [1]. This gap necessitates the use of a memory hierarchy, including a cache, to bridge the performance divide. Caches are designed to store frequently accessed data and allow the processor to access this subset within only a single clock cycle. This enables faster data access latencies, which in turn improve the overall performance and throughput, allowing processors to operate close to their theoretical maximum speed.

However, caches have a wide array of design parameters, and identifying the optimal parameter set is highly application-dependent. This requires a configurable cache generator to find the best cache design parameters for specific application requirements. Therefore, we propose PyCacheGen a highly configurable open-source cache generator implemented in the Python-based Amaranth hardware description language (HDL) [2] which allows for the generation of synthesizable Verilog code. PyCacheGen implements the following features: generation of fully associative, set-associative, and direct-mapped caches, supporting the write-through and write-back policies, with write allocate and no-write allocate policies, an optional write buffer, configurable block size and data width, single-cycle latency for read and write requests, different replacement policies, multiple request ports, flushing, and multiple cache levels. To evaluate the performance, power consumption, energy, and area of the generated caches, we integrated them into the PULPissimo RISC-V SoC platform [3] and synthesized it using GlobalFoundries' 22FDX+ technology node and conducted register-transfer level (RTL) and post-synthesis simulations, executing representative benchmarks from the BEEBS [4] benchmark suite.

CPSWS'25: CPS Summer School PhD Workshop, September 22, 2025, Alghero, Sardinia, Italy

*Corresponding author.

†These authors contributed equally.

0009-0008-9044-0697 (R. Müller); 0000-0002-2701-5881 (K. Lübeck); 0009-0009-9884-9590 (M. Kuhn); 0009-0009-9884-9590 (P. P. Bernardo); 0000-0002-1615-507X (O. Bringmann)

Table 1
Qualitative comparison of PyCacheGen with other configurable open-source caches and generators.

	FPGA Cache Gen. [5]	**VLSI-EDA PoC lib. [6]**	**IOb-Cache [7]**
HDL	Verilog	VHDL	Verilog
Direct-mapped	✓	n/a	✓
Set-associative	(✓) two-way only	✓	✓
Fully associative	✓	n/a	n/a
Replacement policies	Counter / LRU	LRU	LRU / PLRU (tree & MRU)
Write-through policy	✓	✓	✓
Write-back policy	✗	✗	✗
Write buffer	✗	✗	✓
Write allocate	n/a	✗	✗
No-write allocate	n/a	✓	✓
Mult. request ports	✗	✗	✗
License	n/a	Apache License 2.0	MIT License
Eval. technology	Altera Stratix FPGA	Xilinx & Altera FPGA	Xilinx FPGA

	OpenCache [8]	**HPDcache [9]**	**This Work**
HDL	nMigen (Verilog)	SystemVerilog	Amaranth HDL (Verilog)
Direct-mapped	✓	n/a	✓
Set-associative	✓	✓	✓
Fully associative	n/a	n/a	✓
Replacement policies	FIFO / LRU / random	n/a	FIFO / PLRU (tree & MRU) / LRU
Write-through policy	✗	✓	✓
Write-back policy	✓	(✓)[a]	✓
Write buffer	n/a	✓	✓
Write allocate	n/a	n/a	✓
No-write allocate	n/a	n/a	✓
Mult. request ports	✗	✓	✓
License	BSD 3-Clause License	Solderpad HW License v2.1	BSD 3-Clause License
Eval. technology	n/a	(GF 22FDX)[b]	GF 22FDX+

[a]added in public repository, [b]only planned tape-out mentioned

2. Related Work

Several configurable open-source data caches have been proposed. [5] presents a configurable Verilog cache generator for FPGAs that supports fully associative, direct-mapped, and 2-way set-associative caches using the write-through policy with counter-based and LRU replacement policies. The authors provide area and performance results for an Altera Stratix FPGA for all three cache variants.

The VLSI-EDA Pile of Cores (PoC) library [6] offers implementations for various hardware modules in VHDL, enabling the generation of set-associative caches with write-through and no-write allocate policies. The library includes synthesis scripts for different FPGA vendors.

The IOb-Cache [7] is a configurable Verilog cache for direct-mapped and set-associative caches supporting the write-through policy together with a write buffer and no-write allocate. The authors present resource and performance results for Xilinx FPGAs using the Dhrystone benchmark [10].

OpenCache [8] is a Verilog cache generator that utilizes the open-source OpenRAM [11] SRAM compiler for internal cache memory. OpenCache supports generating direct-mapped and set-associative caches that employ the write-back policy.

HPDcache [9] is a configurable data cache designed for RISC-V cores in SystemVerilog, utilizing the write-through policy with a write buffer. It also supports multiple request ports and out-of-order request processing.

While these caches and generators offer a range of open-source implementations, they support either the write-through or write-back policy. In contrast, our configurable cache generator, PyCacheGen, allows for synthesizable multi-port direct-mapped, set-associative, and fully associative caches using either the write-through or write-back policy, along with a configurable write buffer and write or no-write allocation, all within an easy-to-use Python interface. Table 1 provides a detailed comparison of the different open-source caches and generators.

3. PyCacheGen

PyCacheGen is implemented in the Python-based Amaranth HDL, which allows for register-transfer level simulations with the integrated RTL simulator. Additionally, the Amaranth HDL allows for the generation of synthesizable Verilog code for simulation and synthesis with other electronic design automation tools. PyCacheGen is available as open-source under the BSD 3-clause license from `https://github.com/ekut-es/pycachegen`.

To generate a cache hierarchy, an object of the `CacheWrapper` class must be created, which gets passed a list of `CacheConfig` objects containing the parameters of each cache level, which also allow for passing technology-specific memory macros for the caches' internal memories. An example for instantiating a `CacheWrapper` object is presented in Listing 1. Utilizing the Amaranth HDL's `verilog.convert` function, the `cache_wrapper` object is converted into Verilog code. Fig. 1 shows a block diagram of the `CacheWrapper` together with its front and back end interfaces.

To connect the different cache levels to each other and the `CacheWrapper` to the front and back end, the `MemoryBus` is used, which combines several wires into a single bus. The `CacheWrapper` frontend has an additional `hit_o` wire, which is set to high when a request results in a cache hit on the first cache level. To connect the `CacheWrapper` back end to a memory, a memory interface adapter is needed, which translates the `MemoryBus` signals to the signals of the memory interface.

```python
from pycachegen import *
from amaranth.back import verilog

cache_wrapper = CacheWrapper(
    num_ports=1, address_width=25,
    cache_configs=[
        CacheConfig(data_width=32, num_ways=2, num_sets=1, block_size=2,
                    replacement_policy=ReplacementPolicies.PLRU_TREE,
                    write_policy=WritePolicies.WRITE_BACK,
                    write_allocate=True, write_buffer_size=4),
        CacheConfig(data_width=32, num_ways=2, num_sets=2, block_size=4,
                    replacement_policy=ReplacementPolicies.FIFO,
                    write_policy=WritePolicies.WRITE_BACK,
                    write_allocate=True, write_buffer_size=8)],
    main_memory_data_width=128, create_main_memory=False)

with open("cache_wrapper.v", "w") as f:
    f.write(verilog.convert(cache_wrapper, name="CacheWrapper"))
```

Listing 1: Example Python code for generating a cache hierarchy using the `CacheWrapper` class.

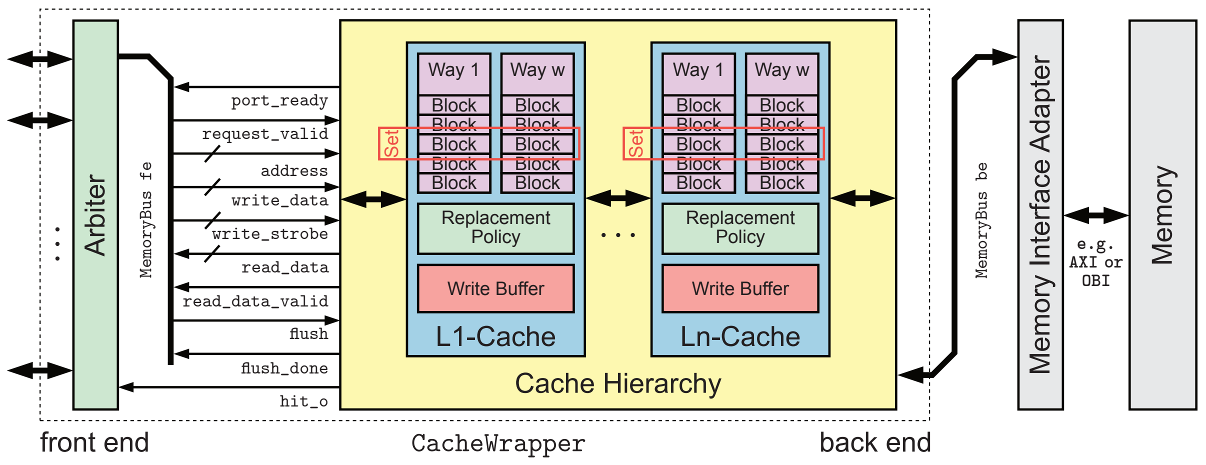
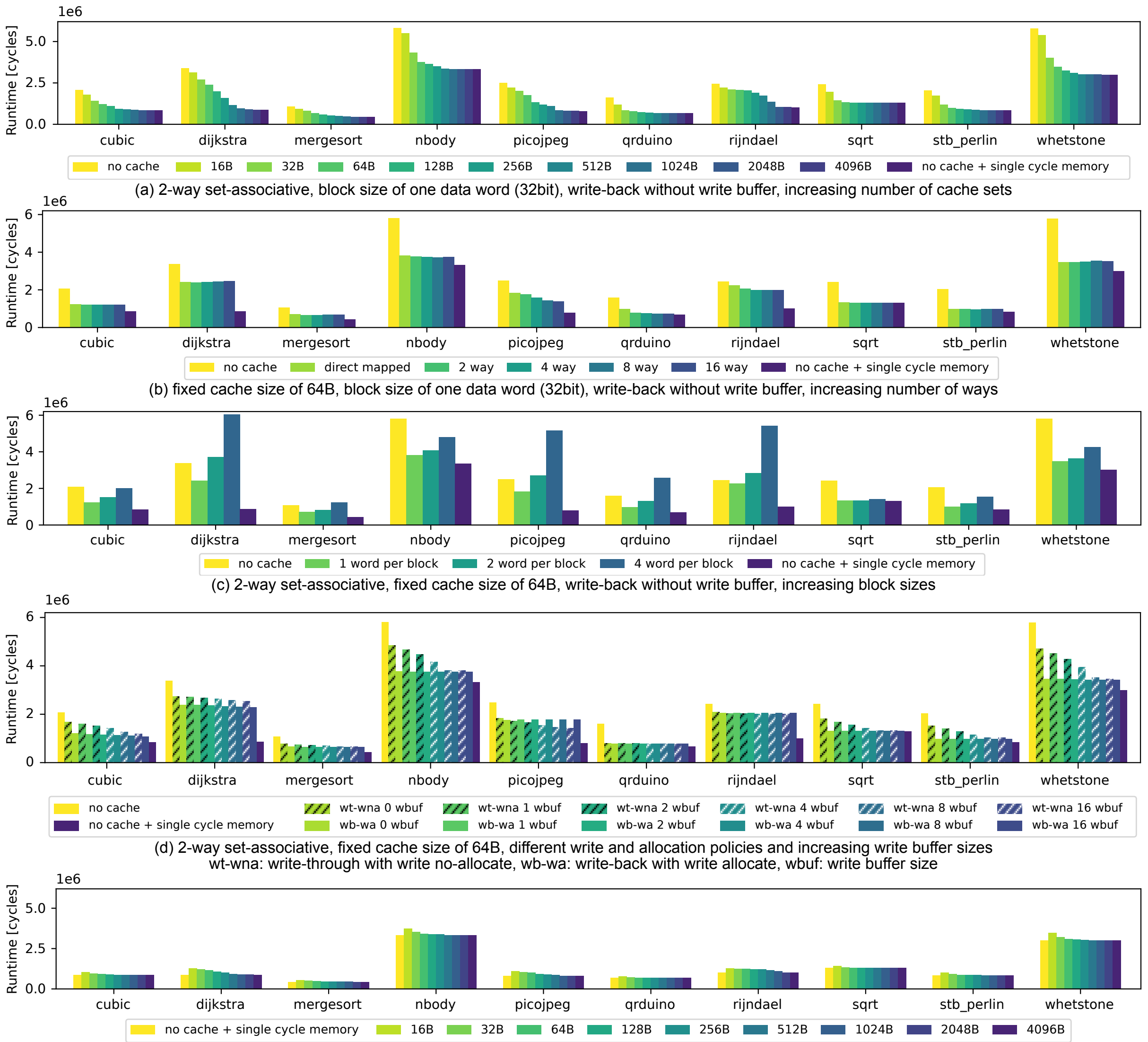


Figure 1: Block diagram of the `CacheWrapper` and its front and back end interfaces.



(a) 2-way set-associative, block size of one data word (32bit), write-back without write buffer, increasing number of cache sets

(b) fixed cache size of 64B, block size of one data word (32bit), write-back without write buffer, increasing number of ways

(c) 2-way set-associative, fixed cache size of 64B, write-back without write buffer, increasing block sizes

(d) 2-way set-associative, fixed cache size of 64B, different write and allocation policies and increasing write buffer sizes
wt-wna: write-through with write no-allocate, wb-wa: write-back with write allocate, wbuf: write buffer size

(e) 2-way set-associative, block size of one data word (32bit), write-back without write buffer, increasing number of cache sets, unit latency memory

Figure 2: Runtime results for different caches integrated into the PULPissimo SoC. For each plot, a specific cache parameter is varied while the other parameters stay fixed.

4. Results

Experimental Setup To evaluate caches generated by PyCacheGen, we integrated the `CacheWrapper` Verilog module as a data cache into the PULPissimo SoC platform [3], positioned between the CV32E40P RISC-V processor and the 128 kB main memory. The main memory was configured with read and write latencies of 8 and 12 clock cycles, respectively, to assess the runtime effects of the data cache for slow memory hierarchies. We also present runtime results using an ideal memory with unit latencies, representing the optimal performance for each benchmark. Runtime results were collected through RTL simulations. For power, energy, and area data, the PULPissimo SoC was synthesized using GlobalFoundries' 22FDX+ technology node at a clock frequency of 200 MHz, utilizing standard cells and low-leakage memory macros from Synopsys. The replacement policy employed was PLRU with a tree data structure. We selected ten benchmarks from the BEEBS suite [4], including cubic, dijkstra, mergesort, nbody, picojpeg, qrduino, rijndael, sqrt, stb_perlin, and whetstone, chosen for their memory reliance and diverse memory access patterns.

Performance Fig. 2 (a) presents the runtime results for a 2-way set-associative cache with a block size of one utilizing the write-back policy and no write buffer. For all benchmarks, the runtime decreases

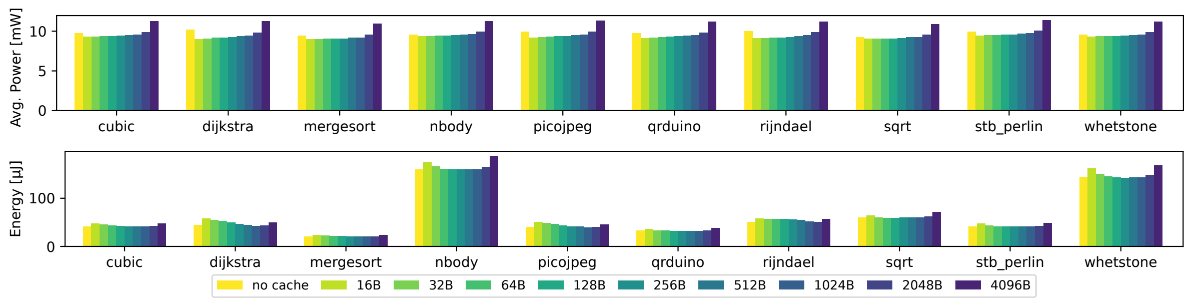


Figure 3: Average power consumption (top) and energy (bottom) for increasing cache sizes for 2-way set-associative caches with a block size of one containing a 32bit data word and using the write-back policy without a write buffer. The cache size is increased by increasing the number of sets.

when the cache size is increased by increasing the number of sets. Especially for caches of size 2048B and 4096 B, the runtime closely matches the best achievable runtime when using no cache and a memory with single-cycle access latencies.

In Fig. 2 (b) runtime results for caches with increasing associativity are shown. All generated caches use the write-back policy with no write buffer and have a size of 64 B, while each block contains one data word. Generally, increasing the associativity has only a minor effect on the runtime.

Fig. 2 (c) depicts the runtime results for 2-way set-associative caches with increasing block sizes and a cache size of 64 B. For all programs, the runtime increases when the block size is increased, while the best runtime is achieved with caches whose block size is set to one. This is because the miss penalty increases with the block size since as many memory read transactions as data words in a block need to be issued to populate one cache block.

A runtime comparison between the write-through and write-back policies with increasing write buffer sizes is shown in Fig. 2 (d). All generated caches are configured as 2-way set-associative, with a size of 64 B and a block size of one, along with increasing write buffer sizes. For each benchmark, hatched and unhatched bars of the same color represent the same write buffer size but different policies. The hatched bars indicate runtime with the write-through policy and no-write allocation, while the unhatched bar to the right uses the write-back policy with write allocation. The write-back policy generally yields better runtime results than the write-through policy for most benchmarks, except picojpeg and rijndael, as it issues memory write transactions only upon block eviction, reducing the memory access frequency. Increasing the write buffer size improves runtime for the write-through policy, as it allows the cache to handle incoming requests without stalling until the write buffer is full. However, for the write-back policy, a larger write buffer does not enhance the runtime, since it issues fewer memory write transactions, leaving the write buffer mostly empty.

Lastly, Fig. 2 (e) shows runtime results when using a memory with single cycle read and write latencies and together with 2-way set-associative caches utilizing the write-back policy and no write buffer with increasing cache sizes. In this case, using a small cache increases the runtime slightly because of frequent conflict misses. Since each memory transaction is routed through the caches, the average access latency is increased compared to using no cache. When increasing the cache size, the runtime almost matches the runtime using no cache across all benchmarks due to a low miss rate.

Power and Energy After presenting runtime results for varying cache parameters, we now provide power and energy measurements from post-synthesis simulations of the entire PULPissimo SoC, including memories and I/O cells with integrated caches of different sizes compared to no cache. The memories are configured with single-cycle access latencies, and the generated caches are 2-way set-associative with a block size of one, using the write-back policy without a write buffer.

Fig. 3 (top) shows the average power for the PULPissimo SoC with and without integrated caches of increasing size. A cache size smaller than 2048 B results in lower average power consumption for all benchmarks due to stalling from frequent cache misses and routing all memory requests through the

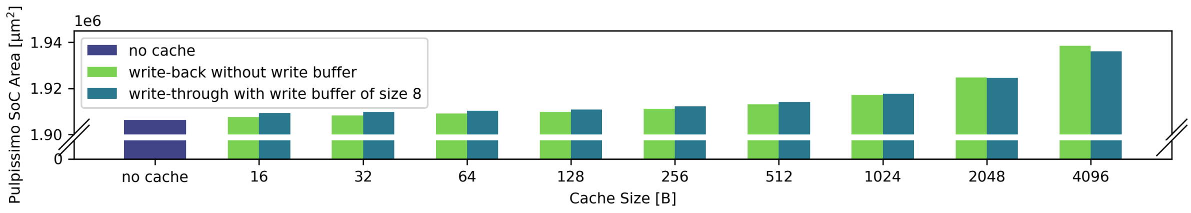


Figure 4: Area results of the PULPissimo SoC with no cache and caches with increasing sizes. The integrated caches are 2-way set associative with a block size of 32 bit containing one data word, either utilizing the write-back policy without a write buffer or the write-through policy with a write buffer of size 8. The cache size is increased by increasing the number of sets.

cache. However, as the cache size increases to 2048 B or 4096 B, average power consumption exceeds that of the no-cache scenario due to larger internal cache memory and reduced stalling.

Fig. 3 (bottom) shows the energy for the PULPissimo SoC with and without integrated cache. Generally, smaller caches lead to higher energy across all benchmarks. However, a 1024 B cache shows a 0.81% lower energy summed across all benchmarks compared to no cache integration, with a mean runtime increase of only 0.93%. Notably, for dijkstra with a 1024 B cache, the energy decreases by 5.51% compared to the no-cache scenario, representing the largest decrease among all benchmarks, with a runtime increase of 2.04%. These energy improvements are attributed to the reduced number of memory accesses facilitated by the cache.

Area Lastly, we present post-synthesis gate-level area results. Fig. 4 shows the whole PULPissimo SoC area including memories and I/O cells with no integrated cache and with caches of different sizes either employing the write-back policy without a write buffer or the write-through policy with a write buffer of size 8. The `CacheWrapper` Verilog module generated by PyCacheGen only slightly increases the area of the PULPissimo SoC with a maximum of 1.69% when using the write-back policy and 1.56% when using the write-through policy with a write buffer. For caches with sizes greater than 1024 B, the area increase, when utilizing the write-back policy, is larger than the area increase when using the write-through policy with a write buffer. This can be attributed to the more complex write-back logic, which scales with the cache size. For a 1024 B cache that implements the write-back policy, which shows the best energy improvements, the area is only increased by 0.57%.

5. Conclusion

This paper introduced PyCacheGen, a highly configurable open-source generator for synthesizable caches. Using the Python-based Amaranth HDL, PyCacheGen generates fully associative, set-associative, and direct-mapped caches that support both write-back and write-through policies, along with an optional write buffer for various replacement and allocation strategies. This flexibility provides more configuration options than existing open-source caches and generators. Our results show that caches generated by PyCacheGen can be integrated into a SoC platform with only a 0.57% area increase while achieving energy reductions of up to 5.51%. RTL and post-synthesis simulations indicate that finding an optimal cache configuration for runtime, power, and energy is highly application-dependent, highlighting the need for a user-friendly and configurable cache generator.

Going forward, we plan to integrate a cache coherency protocol such as MESI to support multicore architectures with multiple caches.

Acknowledgments

This work has been funded by the German Federal Ministry of Research, Technology, and Space (BMFTR) under grant numbers 16ME0129 (Scale4Edge) and 01IS22086H (MANNHEIM-FlexKI).

Declaration on Generative AI

During the preparation of this work, the authors used LanguageTool in order to: Grammar and spelling check, Paraphrase, and reword. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] D. A. Patterson, J. L. Hennessy, Computer Architecture: A Quantitative Approach, 6th edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2019.

[2] Amaranth HDL, 2025. URL: https://github.com/amaranth-lang/amaranth.

[3] P. D. Schiavone, D. Rossi, A. Pullini, A. Di Mauro, F. Conti, L. Benini, Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX, in: 2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S), 2018, pp. 1–3. doi:10.1109/S3S.2018.8640145.

[4] J. Pallister, S. J. Hollis, J. Bennett, BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms abs/1308.5174 (2013). URL: http://arxiv.org/abs/1308.5174. arXiv:1308.5174.

[5] P. Yiannacouras, J. Rose, A Parameterized Automatic Cache Generator for FPGAs, in: Proceedings. 2003 IEEE International Conference on Field-Programmable Technology (FPT) (IEEE Cat. No.03EX798), 2003, pp. 324–327. doi:10.1109/FPT.2003.1275768.

[6] Chair of VLSI Design, Diagnostics and Architecture, PoC - Pile of Cores, 2016. URL: https://github.com/VLSI-EDA/PoC.

[7] J. V. Roque, J. D. Lopes, M. P. Véstias, J. T. de Sousa, IOb-Cache: A High-Performance Configurable Open-Source Cache, Algorithms 14 (2021). doi:10.3390/a14080218.

[8] E. Dogan, H. F. Ugurdag, M. R. Guthaus, OpenCache: An Open-Source OpenRAM Based Cache Generator, 2025. URL: https://github.com/VLSIDA/OpenCache.

[9] C. Fuguet, HPDcache: Open-Source High-Performance L1 Data Cache for RISC-V Cores, in: Proceedings of the 20th ACM International Conference on Computing Frontiers, CF '23, Association for Computing Machinery, New York, NY, USA, 2023, p. 377–378. doi:10.1145/3587135.3591413.

[10] A. R. Weiss, Dhrystone Benchmark, History, Analysis, "Scores" and Recommendations, White Paper, ECL/LLC (2002).

[11] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, M. Sarwar, OpenRAM: An Open-Source Memory Compiler, in: 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2016, pp. 1–6. doi:10.1145/2966986.2980098.