A 2.41-μW/MHz, 437-PE/mm^2 CGRA in 22 nm FD-SOI With RISC-Like Code Generation

Tobias Kaiser and Friedel Gerfers
Chair of Mixed Signal Circuit Design
Technische Universität Berlin, Germany
Email: kaiser@tu-berlin.de

Abstract—**While coarse-grained reconfigurable arrays (CGRAs) have the potential to improve energy efficiency in general-purpose computing beyond the limitations of von Neumann architectures, they suffer from challenges in code generation. Pasithea-1 is a CGRA architecture that aims to combine high energy efficiency with RISC-like programmability. This paper presents its first silicon prototype and a C compiler that uses conventional CPU compiler techniques. Compared to code generation for traditional CGRAs, which require expensive place and route steps, this method of code generation reduces compile times and compiler complexity significantly. Performance and power were measured for a set of benchmark programs written in C. On average, energy efficiency of 195.1 int32 MIPS/mW and active power of 2.41 μW/MHz were achieved. Peak energy efficiency of 558.2 MIPS/mW and peak performance of 97.5 MIPS were measured. Load/store instructions and instruction transfers are identified as critical factors for energy efficiency in Pasithea. In comparison to an MCU with state-of-the-art energy efficiency, Pasithea achieves higher energy efficiency in four of the benchmarked programs. Switched capacitance per benchmark run was reduced by a factor of approximately 1.4, on average. Its 0.75 mm^2 core area and fabric density of 437 PEs/mm^2 enable use in cost-sensitive applications and permit further upscaling.**

I. Introduction

Energy efficiency is a key factor in computing: IoT nodes require improved energy efficiency to accommodate growing workloads and enable operation with smaller power sources. In datacenters, higher energy efficiency translates to cost savings and reduced environmental impact. In the face of thermal throttling, energy efficiency also determines performance.

Coarse-grained reconfigurable array (CGRA) architectures might provide a path beyond the energy efficiency limitations of von Neumann processors. Their array elements are populated with control and data flow primitives of programs (configuration/instructions). Once an array configuration is loaded, hardware units can perform their tasks by exchanging data and control signals. By reusing instructions for multiple executions, CGRAs require no continuous instruction stream. This is in contrast to von Neumann processors, where every instruction execution is preceded by an instruction transfer.

Code generation challenges: While CGRAs have inherent advantages in performance and energy, they are difficult compiler targets in comparison to von Neumann processors [1]. As a result, they have seen little adoption in general-purpose computing. We highlight six key challenges:

1) Instructions are executed based on parallel data flow rather than sequential control flow.
2) Only a limited set of control flow patterns is supported.
3) Per-cycle time-multiplexing of instructions in array elements give rise to scheduling issues.
4) Functionally different array elements are provided for different purposes, introducing mapping and placement problems.
5) Data flow between array elements is constrained, necessitating the use of routing algorithms at compile time.
6) Low-level bitstreams are used for programming, inhibiting portability across CGRA microarchitectures.

The degree to which those aspects apply varies between architectures. RipTide [2] is a recent general-purpose CGRA that emphasizes energy efficiency as well as programmability. It provides primitives for arbitrary control flow (challenge 2) and forgoes per-cycle time-multiplexing (challenge 3). In spite of this, it depends on either SAT solving or integer linear programming for code generation; simple C programs thus take several minutes to compile.

DiAG [3] is a CGRA / von Neumann hybrid that runs RISC-V code instead of a custom bitstream, dynamically constructing hardware datapaths from instruction sequences. This makes code generation easy but leads to a mismatch between hardware and instruction set, imposing penalties in performance, power and area: Dataflow mediated by general-purpose registers is mapped to hardware using area-intensive register lanes. Furthermore, hardware-inherent locality restrictions of data and control flow are not accounted for by the ISA and thus must be hidden by hardware.

In [4], we proposed Pasithea-1, a CGRA with an ISA based on RISC principles optimized for execution in a spatial array.

Contributions of this paper: In this paper, we present the first silicon prototype of Pasithea. While the architecture was previously programmed in an assembly-like language, this paper demonstrates a simple C compiler that uses CPU compiler methods to generate Pasithea machine code. From measurements, performance and energy efficiency is evaluated on program and instruction level. Results are compared state-of-the art CGRAs and an ultra-low-power MCU.

Outline: Section II introduces the Pasithea architecture and links it to the challenges mentioned above. Section III shows the silicon prototype. Section IV describes code generation. Section V introduces the benchmarks used for evaluation.

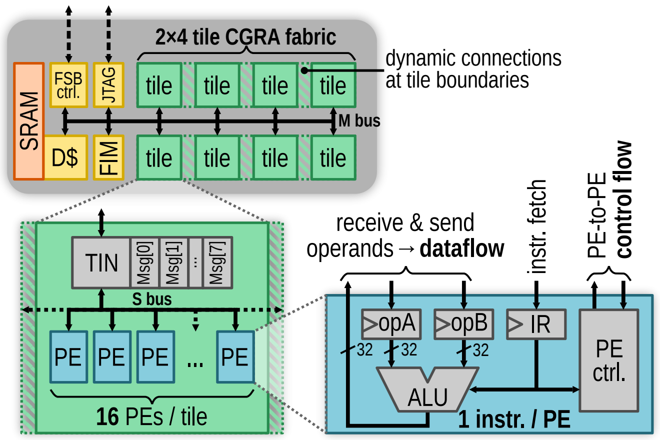


Fig. 1. Pasithea microarchitecture.

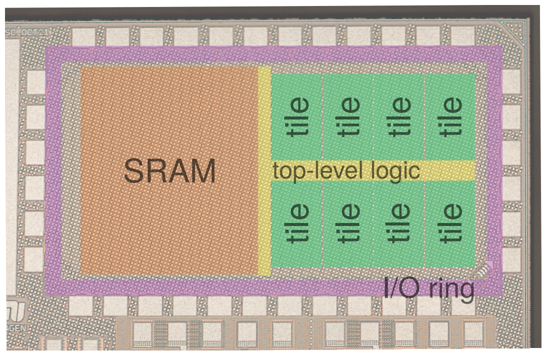


Fig. 2. Die micrograph of the 8-tile CGRA demonstrator with layout view overlay. Area is 1228 µm × 608 µm, excluding I/O ring and pads.

Results are presented in Section VI. Section VII concludes the paper with a discussion.

II. Architecture

This section summarizes the Pasithea-1 architecture, previously described in [4].

Its ISA is based on RISC principles but is designed specifically for execution in a spatial array, in contrast to other RISC ISAs. Compared to typical CGRA architectures, it addresses all six code generation challenges mentioned in Section I. Fig. 1 shows the CGRA microarchitecture. Its fabric comprises 2×4 tiles, each containing 16 processing elements (PEs). All PEs support 32-bit integer operations and are functionally identical, addressing challenge 4. Only one instruction is held per PE, alleviating challenge 3.

Array hardware inherently restricts locality of dataflow and control flow. The ISA exposes this by requiring partitioning of machine code into **fragments** of up to 64 instructions.

PEs obtain inputs from their local operand registers opA and opB. Using the S bus, a PE can send its result to operand registers within the fragment. Up to four **target instruction pointers (TIPs)** per instruction can encode such send operations. This constitutes a static-single use (SSU) encoding (cf. [5]). With TIPs addressing physical registers directly, highly efficient spatial point-to-point dataflow is enabled. This mechanism replaces the central general-purpose register file found in RISC architectures and addresses challenge 5.

Pasithea is driven by sequential control flow rather than parallel data flow, addressing challenge 1. A signal network that is part of the S bus maintains control flow decentrally instead of using a central program counter. Every instruction that produces a result can encode a conditional branch using a special TIP, addressing challenge 2.

Loading code fragments to fabric creates **fragment instances (FIs)**, which encapsulate the execution state of a fragment. Multiple FIs of a single code fragment can coexist at runtime. Fragments are used as machine-level functions. A global entry fragment is instantiated on reset. At runtime, FIs can be created and terminated using special instructions, making function calls possible. After FI termination, machine code is kept in fabric and is managed in an LRU queue. On subsequent function calls, such residual machine code is reused when possible.

The number of FIs that can coexist on fabric is limited ($8 \times$ 16-instr. FIs *or* 2×64-instr. FIs *or* combinations of differently sized FIs). To decouple maximum call stack depth from array capacity, FIs can be transferred between fabric and memory. The current implementation can keep up to 254 FIs in memory. A central FI manager (FIM) supervises FI creation (including instruction fetching), termination and their movement between fabric and memory. It also determines when FIs need to be moved from fabric to memory (evicted) or from memory to fabric (restored) based on an LRU queue of idle FIs in fabric and a second queue of FIs in memory that are ready to resume processing.

FIs exchange messages via the M bus using their tile interface nodes (TINs), e.g. for argument and return value passing. A TIN can receive and buffer up to eight differently tagged 32-bit message words. Unique FI addresses (FIAs) are maintained in hardware throughout FI lifetime. Coexisting FIs are concurrent; message passing is therefore also needed for event ordering, e.g. to wait for subroutine completion. FIs enable reentrant subroutines and inter-fragment control and data flow. This complements intra-fragment control and data flow based on TIPs.

III. Silicon Prototype

Pasithea-1 was implemented in GlobalFoundries 22 nm FD-SOI CMOS [6]. Fig. 2 shows the prototype die.

Due to its high ratio of logic area to switching activity, ultra-low-leakage (ULL) standard cells with 28 nm gate biasing were used. Its $0.75\,\mathrm{mm}^2$ core includes 256 kB SRAM for program and data and a 4×32B data cache. To reduce switching activity at its 34.2k flip-flops, the design comprises 500 clock gates inserted at different levels in the hierarchy. The tiles were implemented as physically identical instances ($235 \times 141\,\mu\mathrm{m}$). Including top-level logic overhead, a fabric density of 437 $\mathrm{PEs/mm}^2$ was achieved.

A Xilinx Artix 7 FPGA was used for interfacing and clock generation. The core voltage V_{DD} was supplied by a

Keithley 236 source measure unit, enabling accurate current measurements of the system.

IV. Code Generation

To show how code generation is facilitated by the Pasithea-1 ISA, a C compiler based on the LLVM compiler infrastructure [7] (version 15.0.6) was implemented.

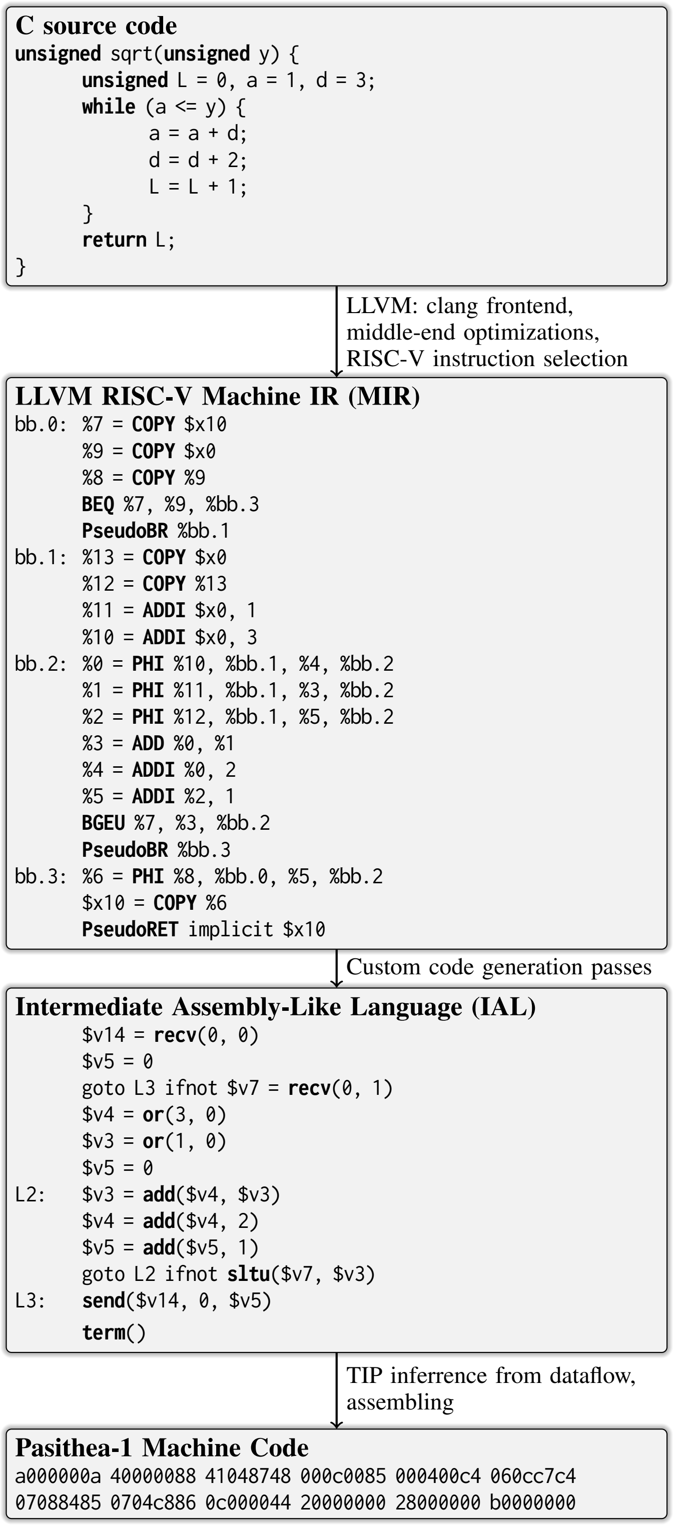
clang translates C code to LLVM IR, which is then further optimized by the LLVM middle-end (*opt*). Using LLVM's SelectionDAG module, preliminary instruction selection is then performed for 32-bit RISC-V [8], on which Pasithea's instruction set is based. The SSA-form machine intermediate representation (MIR) is then further processed by a series of compiler passes implemented in Python:

1) Where required, RISC-V opcodes are translated to Pasithea opcodes; branch instructions are replaced with regular instructions and special branch TIPs.
2) Offset additions that are part of load and store instructions are moved into separate `add` instructions.
3) Function calls are lowered.
4) Argument reads and return value writes are lowered.
5) SSA is eliminated.
6) Constant and copy propagation are performed.
7) Immediate values are simplified and legalized.
8) Where possible, `add` instructions are combined with subsequent load or store instructions.
9) Assignments and branches are combined where possible.
10) Dead code and redundant branches are eliminated.

Transformations are based on dataflow analysis. The resulting intermediate assembly-like language (IAL) and its translation to Pasithea machine code was previously described in [4]: Based on another step of dataflow analysis, IAL virtual register assignments are translated to per-instruction sets of dataflow TIPs: one TIP for each valid use following a virtual register assignment. Up to four TIPs can be encoded per instruction. If this does not suffice, additional copy operations (opcode `or`) are inserted. Machine code is then assembled.

Fig. 3 shows an example of the described code generation process for a simple integer square root function. The RISC-V MIR code contains copy instructions between virtual and physical registers at places where physical registers are fixed by the calling convention. Steps 3 (not present in the example) and 4 eliminate all uses of RISC-V physical registers and adapt the code to Pasithea's calling convention, which is described in [4]. The instruction **`recv`**`(0, 1)` receives the first function argument by reading message register 1. The return value `$v5` is passed back to the caller by the instruction **`send`**`($v14, 0, $v5)`. `$v14` is the caller's FI address. It is used as a message destination address for the return value. `goto L3 ifnot $v7 =` **`recv`**`(0, 1)` is an example where step 9 combined a conditional branch and an assignment to a single instruction.

It is currently required that C functions / MIR functions are small enough for the resulting Pasithea fragment not to exceed the 64-instruction limit. To support arbitrary function sizes, an additional pass would have to partition large functions into multiple fragments, minimizing inter-fragment data and


C source code
LLVM: clang frontend, middle-end optimizations, RISC-V instruction selection
LLVM RISC-V Machine IR (MIR)
Custom code generation passes
Intermediate Assembly-Like Language (IAL)
TIP inferrence from dataflow, assembling
Pasithea-1 Machine Code

C source code

```
unsigned sqrt(unsigned y) {
      unsigned L = 0, a = 1, d = 3;
      while (a <= y) {
            a = a + d;
            d = d + 2;
            L = L + 1;
      }
      return L;
}
```

LLVM RISC-V Machine IR (MIR)

```
bb.0: %7 = COPY $x10
      %9 = COPY $x0
      %8 = COPY %9
      BEQ %7, %9, %bb.3
      PseudoBR %bb.1
bb.1: %13 = COPY $x0
      %12 = COPY %13
      %11 = ADDI $x0, 1
      %10 = ADDI $x0, 3
bb.2: %0 = PHI %10, %bb.1, %4, %bb.2
      %1 = PHI %11, %bb.1, %3, %bb.2
      %2 = PHI %12, %bb.1, %5, %bb.2
      %3 = ADD %0, %1
      %4 = ADDI %0, 2
      %5 = ADDI %2, 1
      BGEU %7, %3, %bb.2
      PseudoBR %bb.3
bb.3: %6 = PHI %8, %bb.0, %5, %bb.2
      $x10 = COPY %6
      PseudoRET implicit $x10
```

Intermediate Assembly-Like Language (IAL)

```
      $v14 = recv(0, 0)
      $v5 = 0
      goto L3 ifnot $v7 = recv(0, 1)
      $v4 = or(3, 0)
      $v3 = or(1, 0)
      $v5 = 0
L2:   $v3 = add($v4, $v3)
      $v4 = add($v4, 2)
      $v5 = add($v5, 1)
      goto L2 ifnot sltu($v7, $v3)
L3:   send($v14, 0, $v5)
      term()
```

Pasithea-1 Machine Code

```
a000000a 40000088 41048748 000c0085 000400c4 060cc7c4
07088485 0704c886 0c000044 20000000 28000000 b0000000
```

Fig. 3. Compiling example `sqrt` function from C to CGRA machine code.

control flow. This is known as temporal partitioning in CGRA compilers [1] and is also employed in CPU compilers to manage instruction locality and improve code generation [9].

Another limitation is that C local variables must currently reside in operand registers and can not be allocated on stack. This could be addressed by either introducing a stack pointer, passed as additional message between fragments, or by using heap allocation for this purpose.

V. Benchmarks

The following benchmark programs were successfully compiled from C code and executed on the Pasithea CGRA: *euclid* (Euclid's gcd algorithm), *stein* (Stein's binary gcd algorithm), *crc32* (cyclic redundancy check), *md5* (message digest 5), *sqrt* (integer square root algorithm shown in Fig. 3), *dijkstra* (shortest path search), *tsort* (tree sort) and *qsort* (quicksort).

The chosen benchmarks vary in code complexity, locality of dataflow, the degree to which nested subroutine calls are used and the number and locality of loads and stores. Algorithms for which multiplication or division is critical were not included, as the current prototype does not provide hardware support for those operations.

The average compile time was 2.1 s across benchmark programs, which is significantly faster than RipTide's compiler.

SleepRunner [10], a recent ARM Cortex-M0 based MCU with state-of-the-art energy efficiency, was selected as benchmarking reference. To minimize instruction fetch energy, it uses an ultra-low-power 32-kB 8T SRAM macro, which reduces access energy 5× compared to a foundry-provided SRAM of equal size. We compiled our benchmarks for ARM Cortex-M0 using GCC (version 12.2.1 with `-O2`) and obtained cycle counts by running them on a commercial MCU. The cycle counts were then used to estimate energy and performance of the benchmarks on SleepRunner.

A direct comparison with state-of-the-art CGRAs such as [2], [11], [12] was not possible, as they were evaluated using a different set of benchmarks.

VI. Results

The silicon prototype was verified by running a small test program at different core supply voltages and clock frequencies to verify correct functioning of memory and CGRA fabric. The test program was executed successfully over a wide V_{DD} range, from 0.42 V to 0.9 V. Fig. 4 shows maximum operating frequency and energy use over this supply voltage range. The minimum energy point (MEP) was found at $V_{DD} = 0.58\,V$ and $f_{clk} = 5.9\,MHz$.

A. Benchmark Execution

The upper half of Table I shows measurement results for the benchmark programs at MEP. Measurements were taken during a continuous loop of program invocations. Residual code fragments from prior loop iterations are reused to the degree to which the architecture implements this.

The programs euclid, stein, sqrt, crc32 and tsort fit into fabric, resulting in zero instruction transfers after the first outermost loop iteration. For dijkstra and md5, not all sequentially

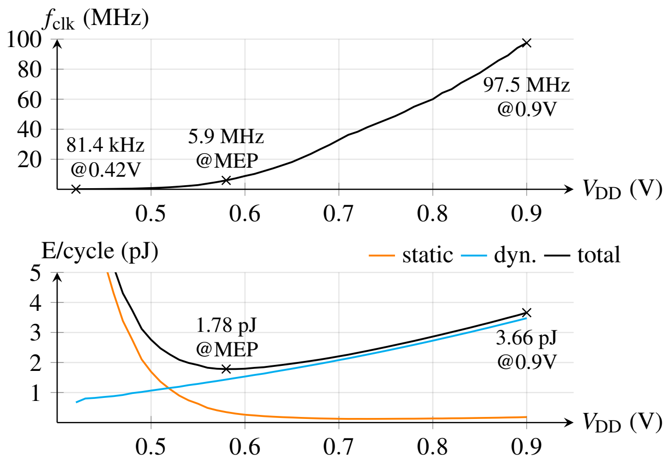


Fig. 4. Maximum operating frequency and energy per cycle over V_{DD} for test program execution at room temperature.

TABLE I
Measurement results for program execution on Pasithea CGRA at MEP and machine code size comparison.

Program	µW/MHz	CPI	MIPS/mW	Code Size (B)	
	this work				CM0
euclid	1.78	1.57	357.2	208	96
stein	1.97	1.47	345.7	280	152
sqrt	1.79	1.00	558.2	84	54
crc32	2.48	1.33	303.8	136	88
md5	3.48	2.07	139.0	760	492
dijkstra	2.91	3.43	100.2	980	616
tsort	2.75	3.01	120.7	400	252
qsort	2.64	6.37	59.6	280	112
geom. mean	**2.41**	**2.12**	**195.1**		
nops / 1 tile	0.97	1	1025.7		
counter / 1 tile	1.39	1	719.3		
counter / 2 tiles	1.73	1	579.6		
maxtoggle / 1 tile	2.53	1	395.9		
maxtoggle / 2 tiles	3.98	1	251.4		
load hit	2.64	8	46.6		
store hit	3.64	4	68.7		
load miss	4.74	9	23.1		
store miss	9.69	4 – 6	17.9		

invoked code fragments could be held in fabric at the same time; due to this, parts of the fabric must be reconfigured on every program invocation, i.e. some instruction fetching occurs. The performance (CPI) and energy penalty of repeated instruction fetching is apparent in Table I.

qsort quickly exceeds the fabric's call stack capacity by recursively spawning FIs. This triggers expensive evict/restore operations, which are absent in the other benchmarks. A steep reduction of CPI and energy efficiency is the result.

The amount and locality of load/store operations also im-

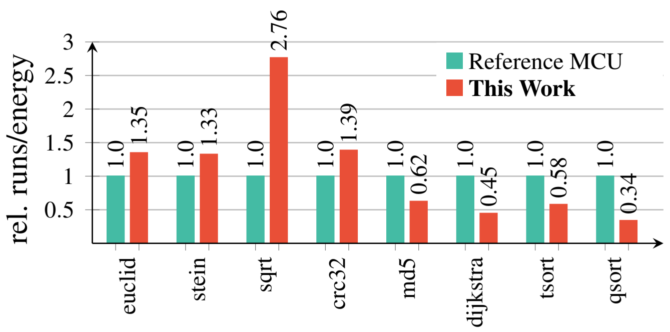


Fig. 5. Comparison of energy efficiency (runs/energy) between Pasithea measurements and reference MCU (SleepRunner) estimates.

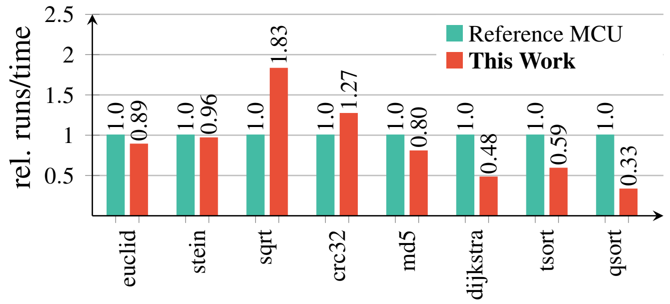


Fig. 6. Comparison of performance (runs/time) between Pasithea measurements and reference MCU (SleepRunner) estimates.

pacts performance and energy use significantly. euclid, stein and sqrt perform no loads or stores. crc32 loads a byte array in sequence, which increases energy per cycle slightly. dijkstra, tsort and qsort operate on data structures in memory and thus have high load/store frequencies. This degrades their energy efficiency and performance. The same is true for the md5 implementation, whose inner loop utilizes a LUT in memory.

B. Single-Instruction Measurement

To identify limiting factors, we measured performance and energy for key instructions using special loop test cases and differential power analysis. Initial instruction fetching is excluded. Results are shown in the lower half of Table I.

The nop instructions only pass control to the next PE in sequence and encode no data transfers via the S bus. Their energy efficiency of $1025.7\,\mathrm{MIPS/mW}$ can be interpreted as the architecture's upper efficiency bound. In the *counter* test, some nops were replaced by incrementing add instructions that transmit their results using the S bus. Table I shows the corresponding increase of energy use for an individual add instruction. The *maxtoggle* test measures a xor instruction toggling all input operand bits. This causes energy to rise substantially, showing the impact of data-dependent S bus activity on energy efficiency.

Furthermore, energy use and performance of load/store instructions were measured. Table I shows that loads and stores are highly expensive in terms of energy use and performance in the implemented architecture. This is due to two factors: Firstly, all load and store requests are forwarded from originating PEs through S bus, TIN and M bus to the data cache. This requires time and switching energy. Secondly, with $18\,\mathrm{pJ}$ for a 256-bit read and $21\,\mathrm{pJ}$ for a write access, SRAM access cycles use $\sim 8\times$ more energy than clock cycles without SRAM access.

C. Static Power Dissipation

At $V_{\mathrm{DD}} = 0.8\,\mathrm{V}$, a static power of $8.16\,\mathrm{\mu W}$ was measured. At $V_{\mathrm{DD}} = 0.58\,\mathrm{V}$ (MEP), measured static power was $2.06\,\mathrm{\mu W}$. According to characterization data, $58\,\%$ of static power is dissipated by the SRAM macro.

D. Comparison with MCU

Figs. 5 and 6 compare energy efficiency and performance between measurement results from Pasithea and SleepRunner estimations.

Fig. 5 shows that Pasithea exceeds SleepRunner in efficiency when only few memory accesses are performed and instruction reuse manages to keep the number of instruction fetches small. In cases where memory accesses are frequent (md5, dijkstra, tsort, qsort), performance drops at least slightly below that of the reference MCU. For qsort, FI evict/restore operations aggravate this drop in efficiency.

The same effects also impact performance, as shown in Fig. 6. In cases where memory access frequency is low, relative energy efficiency is higher than the corresponding relative performance.

For sqrt, the large differences in performance and energy between Pasithea and SleepRunner are due to GCC producing suboptimal ARM code for that particular program (two branches per loop iteration).

The code sizes in Table I show that Pasithea machine code is approximately $1.76\times$ larger than ARM Cortex-M0 machine code, which achieves extraordinary high code densities by exclusively using 16-bit instruction encoding. Pasithea, on the other hand, uses 32-bit instructions that are extended by up to two 32-bit prefix words for greater immediate ranges and additional TIPs. Fragment start and end markers, and unused TIP and immediate fields contribute to Pasithea's lower code density. We speculate that dataflow encoding using TIPs is also inherently less compact than dataflow encoding using GPRs.

VII. Discussion

Table II compares Pasithea-1 with SleepRunner and three state-of-the-art CGRAs. For peak performance and peak energy efficiency, the sqrt benchmark results were selected.

Comparison to MCU: In contrast to SleepRunner, Pasithea offers $4\times$ more memory, does not use energy-optimized data memory, runs from a single core supply voltage and does not utilize low-power techniques such as back-gate biasing. Despite those factors, Pasithea surpasses or matches SleepRunner in energy efficiency and performance in many cases, as shown in Figs. 5 and 6. When normalizing energy to ${V_{\mathrm{DD}}}^2$ (in case of SleepRunner, for its three supply voltages respectively), a $1.4\times$ average ($4.4\times$ peak) reduction of switching activity (capacitance) per benchmark run becomes apparent.

TABLE II
COMPARISON OF RESULTS TO ULP CGRAS AND RISC MCU

	SleepRunner [10]	**HyCube [11]**	**Amber [12]**	**RipTide [2]**	**Pasithea-1 (this work)**
Node	28 nm FD-SOI	40 nm LP	16 nm FinFET	22 nm FinFET	22 nm FD-SOI
Results	silicon	silicon	silicon	post-syn.	silicon
Type	RISC MCU	CGRA			
		MCU-supervised			self-managed
ISA	ARM CM0	custom low-level bitstream			RISC-like
$f_{\text{clk,max}}$	80 MHz	753 MHz	955 MHz	50 MHz	97 MHz
V_{DD} (V)	0.8/0.5/0.4	0.8 – 1.1	0.84 – 1.29	unknown	0.42 – 0.9
Fabric Size	N/A	4×4 PEs	384 PEs	6×6 PEs	8×16 PEs
PEs/mm^2		5.6	19.1	144	437
µW/MHz	3.3	84.4	unknown	14.8	1.79
Total mm^2	0.574	2.87	20.1	0.25	0.75
Memory	64 kB	4 kB	4608 kB	256 kB	256 kB
Benchmark	Dhrystone	FFT	dense lin. algebra		sqrt
Peak Performance	100 DMIPS	5380 MOPS	367 int16 GOPS	164 MOPS	97.5 int32 MIPS
Peak Efficieny	385 DMIPS/mW	26.4 int32 MOPS/mW	538 int16 MOPS/mW	180 MOPS/mW	558.2 int32 MIPS/mW
Static Power	8.4 µW	unknown	unknown	< 9.6 µW	2.06 µW

The used ULL transistors afford Pasithea a $4\times$ lower static power with full data retention compared to SleepRunner. This shows that the presented architecture is suitable for applications that rely on low energy use during standby operation.

Comparison to CGRAs: Traditionally, CGRAs have been optimized to outperform von Neumann architectures in throughput, which can be seen in the performance figures reported for Amber and HyCube. In contrast to this, RipTide and Pasithea emphasize energy efficiency and programmability, which we consider key challenges in contemporary computer architecture. Compared to RipTide, Pasithea reduces compile times from minutes to seconds. The performance and efficiency figures listed in Table II give a rough overview, but do not allow a precise comparison due to differences in what instructions or operations were run and how they were counted. DiAG [3], whose instruction reuse approach has similarities to the presented architecture, was not included in Table II, as no energy efficiency data suitable for a comparison was reported.

Conclusion: We have shown that, by reducing instruction fetching, the presented architecture can reduce switching activity and improve energy efficiency in comparison to state-of-the-art MCUs. Based on the presented findings, load/store operations were identified as bottlenecks and candidates for further optimization. The compiler described in Section IV demonstrated code generation for Pasithea's RISC-like instruction set. By addressing the code generation challenges from Section I, the architecture enables RISC-like code generation and fast compile times. This opens the door for further research towards future adoption of CGRAs in general-purpose computing.

ACKNOWLEDGEMENTS

We thank GlobalFoundries for 22 nm FD-SOI technology access and EUROPRACTICE for providing design tools.

REFERENCES

[1] J. M. P. Cardoso *et al.*, "Compiling for reconfigurable computing," *ACM Computing Surveys*, vol. 42, no. 4, pp. 1–65, Jun. 2010.

[2] G. Gobieski *et al.*, "RipTide: A programmable, energy-minimal dataflow compiler and architecture," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, Oct. 2022.

[3] D. K. Wang and N. S. Kim, "DiAG: A dataflow-inspired architecture for general-purpose processors," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, Apr. 2021.

[4] T. Kaiser and F. Gerfers, "Pasithea-1: An energy-efficient self-contained CGRA with RISC-like ISA," in *Architecture of Computing Systems*, Springer International Publishing, 2022, pp. 33–47.

[5] R. Lo *et al.*, "Register promotion by sparse partial redundancy elimination of loads and stores," *ACM SIGPLAN Notices*, vol. 33, no. 5, pp. 26–37, May 1998.

[6] R. Carter *et al.*, "22nm FDSOI technology for emerging mobile, internet-of-things, and RF applications," in *2016 IEEE International Electron Devices Meeting (IEDM)*, IEEE, Dec. 2016.

[7] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization*, IEEE, 2004.

[8] A. Waterman and K. Asanović, "The RISC-V instruction set manual, volume I: User-level ISA, document version 2019121," RISC-V Foundation, Tech. Rep., 2019.

[9] P. Zhao and J. N. Amaral, "Ablego: A function outlining and partial inlining framework," *Software: Practice and Experience*, vol. 37, no. 5, pp. 465–491, 2007.

[10] D. Bol *et al.*, "SleepRunner: A 28-nm FDSOI ULP Cortex-M0 MCU with ULL SRAM and UFBR PVT compensation for 2.6–3.6-uW/DMIPS 40–80-MHz active mode and 131-nW/kB fully retentive deep-sleep mode," *IEEE Journal of Solid-State Circuits*, vol. 56, no. 7, pp. 2256–2269, Jul. 2021.

[11] B. Wang *et al.*, "HyCUBE: A 0.9 V 26.4 MOPS/mW, 290 pJ/op, power efficient accelerator for IoT applications," in *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, IEEE, Nov. 2019.

[12] A. Carsello *et al.*, "Amber: A 367 GOPS, 538 GOPS/W 16nm SoC with a coarse-grained reconfigurable array for flexible acceleration of dense linear algebra," in *2022 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*, IEEE, Jun. 2022.