PELS: A Lightweight and Flexible Peripheral Event Linking System for Ultra-Low Power IoT Processors

Alessandro Ottaviano*[x], Robert Balas*[x], Philippe Sauter*, Manuel Eggimann*, Luca Benini*†
* *Integrated Systems Laboratory, ETH Zurich*, Switzerland
† *Department of Electrical, Electronic, and Information Engineering, University of Bologna*, Italy

***Abstract*—A key challenge for ultra-low-power (ULP) devices is handling peripheral linking, where the main central processing unit (CPU) periodically mediates the interaction among multiple peripherals following wake-up events. Current solutions address this problem by either integrating event interconnects that route single-wire event lines among peripherals or by general-purpose I/O processors, with a strong trade-off between the latency, efficiency of the former, and the flexibility of the latter. In this paper, we present an open-source, peripheral-agnostic, lightweight, and flexible Peripheral Event Linking System (PELS) that combines dedicated event routing with a tiny I/O processor. With the proposed approach, the power consumption of a linking event is reduced by 2.5 times compared to a baseline relying on the main core for the event-linking process, at a low area of just 7 kGE in its minimal configuration, when integrated into a ULP RISC-V IoT processor.**

***Index Terms*—peripheral, event linking, low-power, real-time, predictability, RISC-V**

I. Introduction

The combination of ultra-low-power (ULP) and performance in the Internet of Things (IoT) market segment leads to demanding *energy-efficiency* requirements for *battery-operated* devices [1], where power savings prolong device autonomy. Similarly, in *always-on* wearables such as those found in augmented reality visual processing [2], monitoring services [3], and surveillance cameras [4] energy-efficiency directly translates to reduced cost, and extended device lifetime. Furthermore, emerging application domains such as in-vehicle communication for automotive demand high computation-per-watt [5] as well as real-time and predictable interaction with a wide range of peripherals and sensors [6], [7] to achieve optimal and fine-grained vehicle control [8].

To meet the energy requirements of such applications, systems on chip (SoCs) are traditionally designed to reduce standby leakage power when *idle*. This is achieved by enclosing the *processing domain* and the *I/O domain* in different power regions selectively powered on (*active mode*) or off (*sleep mode*) [9]. The *processing domain* only wakes up when a specific condition is detected by the surrounding sensors [10]. For such reasons, a typical design goal of ULP systems is to minimize the number of wake-up events for the *processing domain*. This not only saves power but also improves the system's predictability by lowering jitter and latency.

A key challenge for ultra-low-power (ULP) devices is handling peripheral linking. Usually, the processing domain handles peripheral interactions following wake-up events. To reduce the number of wake-up events in the processing domain, peripherals can be made more autonomous by enabling inter-peripheral communication (*peripheral event linking*). This communication channel lets peripherals interact directly, potentially bypassing the processing domain. Peripheral linking can be realized through single-wire event lines (*instant actions*) or with arbitrary commands involving memory-mapped accesses to the peripheral over the system interconnect (hereafter, *sequenced actions*). By circumventing the central processing unit (CPU) and the system interconnect instant actions reduce access latency and minimize jitter for real-time applications.

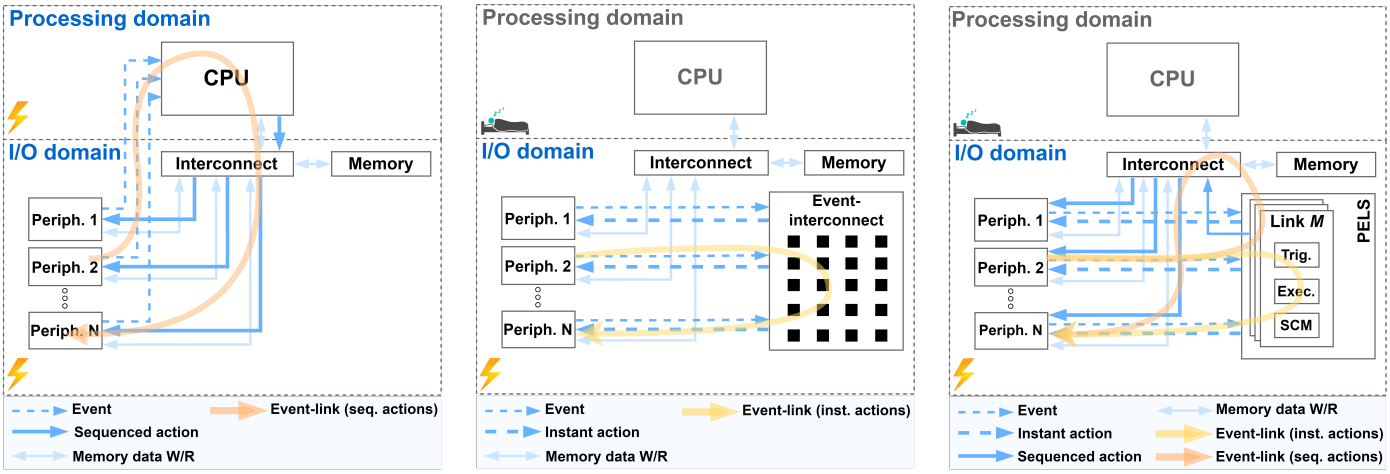
For example, consider a periodic timer overflow triggering an analog-to-digital converter (ADC) conversion or an SPI end-of-transfer event starting a new data transfer. In a traditional system, I/O direct memory access (DMA) engines [11] can be used for autonomous handling of sensor data movement but are not sufficient to orchestrate inter-peripheral linking, which still requires the main CPU to wake up and intervene at each linking event following a classic interrupt mechanism, as depicted in Figure 1a, adversely affecting the system's response latency, predictability, and power consumption.

Autonomous inter-peripheral event-linking units have been proposed in literature and industry as a complementary solution to I/O DMA engines to achieve full decoupling between the *processing domain* and the *I/O domain*. Existing solutions rely on a peripheral-event interconnect with configurable connections and minimal processing capabilities (Section II-B and Section II-C), as depicted in Figure 1b. Although area-efficient and low-latency, these approaches require peripheral-aware design to provide a fixed, built-in set of actions. Other approaches employ I/O processors in the *I/O domain*. They provide the highest flexibility but incur a higher cost in terms of response latency due to the need to access power-hungry static random access memory (SRAM) banks within the SoC through the system bus (Section II-C2). To the best of the authors' knowledge, a minimally intrusive, low-power, and configurable solution integrating the best of both worlds is missing in the research and industry landscapes.

Contribution To address these limitations, we propose PELS, a flexible, lightweight, and open-source event-linking unit. The contributions of this work are:

[x] Both authors contributed equally to this research.

979-8-3503-4859-0/DATE24/© 2024 EDAA



(a) **(b)** **(c)**

Fig. 1. Profiles of different solutions for handling inter-peripheral communication in ULP SoCs. **(a)** Traditional software interrupt approach. The main CPU is woken up at each inter-peripheral event. **(b)** Configurable event-interconnect approach with minimal processing capability. The processing domain is bypassed and can sleep, while the event interconnect assures low latency but requires joint codesign with the peripheral system, affecting flexibility. **(c)** Proposed PELS design that combines single-wire event lines and sequenced actions through the interconnect. PELS doesn't require a priori knowledge of the peripheral system and trades off the low latency and flexibility of existing approaches. (Section IV).

- We design a microcode-based architecture that supports both (i) *instant actions*, i.e., single-wire event lines connecting a *producer* to a *consumer* peripheral, and (ii) *sequenced actions*, i.e., arbitrary commands to control the peripherals through the system interconnect, combining the advantages of existing solutions (Section III). Contrary to typical fully-fledged general-purpose processors, PELS hosts a tiny standard cell memory (SCM) to reduce instruction fetch latency and power envelope through the system interconnect bus (Section IV).
- To the best of the authors' knowledge, PELS is the first event-linking system available open-source [1] and integrated into a microcontroller unit (MCU)-class, open-source, RISC-V SoC [12] targeting ULP IoT applications.
- We provide a detailed performance and cost assessment of PELS in terms of power consumption and area (Section IV). We compare it with a baseline using a traditional interrupt-based mechanism that relies on the main processing core for the event-linking. Furthermore, we show that PELS remains competitive with state-of-the-art (SOTA).

II. Related Works

In this section, we describe the main techniques for autonomous peripheral-event handling. We classify the SOTA into three categories. Table I summarizes the main solutions from industry and academia with a feature-based comparison.

A. Software Interrupts

In a purely software (SW)-driven approach, peripheral events are handled and routed through the processing domain using common interrupt handlers (Figure 1a). This solution creates dependencies between the *I/O domain* and *processing domain*, which can increase the response latency and power consumption due to the CPU waking up to handle the event processing.

B. Peripheral-event interconnect

A common solution to handle inter-peripheral linking in hardware (HW) uses peripheral-event interconnects. In their simplest form, these are a set of *fixed* connections between peripherals to forward events to each other. A significantly more flexible solution is to make the peripheral-event interconnect routing configurable. Several routing topologies exist, such as matrix configurations [17] or multiple multiplexer-demultiplexer channels [13] [14] [15] [16]. Figure 1b shows a representation of such systems. While these approaches offer good area and power efficiency, programmability, and predictable as well as low event latency, they lack flexibility as inter-peripheral connections are decided at design time, i.e., the peripheral design has to be tailored to the event interconnect.

C. Processing on interconnect

1) FPGA approach: If an application demands complex criteria to trigger an action, the event-interconnect topology might be insufficient, as it still requires the processing system to intervene. To address this limitation, the event linking system can be enhanced with minimal processing capabilities, as in the case of the Peripheral Reflex System [13], or with configurable logic blocks (CLBs) of arbitrary complexity de facto integrating a small field-programmable gate array (FPGA) within the peripheral system. A common approach is adding a CLB to the channel topology with minimal inputs and outputs that can also be connected to another channel [14] [19].

2) I/O processor: The last approach in this category consists of adding a tiny general-purpose processor into the peripheral system, which can directly access other peripheral devices through the system interconnect in the *I/O domain* as the

[1] https://www.github.com/pulp-platform/pels

TABLE I
FEATURE COMPARISON OF EXISTING SOLUTIONS FOR AUTONOMOUS PERIPHERAL-EVENT HANDLING. WE FURTHER HIGHLIGHT WHETHER THE ANALYZED SOLUTION IS AVAILABLE IN THE OPEN-SOURCE DOMAIN.

System	Event Routing topology	Event Processing	Instant Actions	Sequenced Actions	Open source
Industry					
Silicon Labs PRS [13]	Channel	Combinational logic	✓	✗	✗
Renesas LELC [14]	Channel	CLB	✓	✗	✗
Microchip EVSYS [15]	Channel	Custom [a]	✓	✗	✗
Nordic PPI [16]	Channel	Custom [b]	✓	✗	✗
STMicroelectronics PIM [17]	Matrix	✗	✓	✗	✗
NXP XGATE [18]	—	Microcode [c]	✗	✓	✗
Academia					
AESRN (Bjørnerud et al. [19])	Channel	CLB [d]	✓	✗	✗
This work	Channel	Microcode	✓	✓	✓

[a] Up to three events can be routed to the Configurable Custom Logic (CCL), which allows the generation of a new output event as a function of the input events. The function is given by a look-up table (LUT).
[b] One channel can trigger up to two actions simultaneously, which can be seen as limited broadcasting.
[c] Designed as a co-processor to take the interrupt load off the main processor.
[d] Uses fully asynchronous logic for processing and routing

main core in the *processing domain* would [18]. The latency of solutions based on a general-purpose I/O processor can be limited by (i) the *system interconnect*, as a slow bus affects the latency while the routing arbitration influences the system's predictability; (ii) the *memory*, as using the system's local memory trades off area reuse for latency; (iii) *datapath width*, which should be at least equal to the system interconnect width to complete accesses within one clock cycle.

Compared to SOTA, PELS allows full programmability of the linking topology with two operation modes. On the one hand, it allows to trigger instant actions when strict latency constraints are required. On the other hand, its microcode design enables flexible integration of new sequenced actions for peripherals as long as they are realizable through the system interconnect. Furthermore, a priori knowledge of the peripheral system design is not required.

III. ARCHITECTURE

This section describes PELS's microarchitecture, depicted in Figure 2. PELS collects events from peripherals and triggers the execution of a sequence of *commands* (Section III-2), referred to as a *sequenced action*. Hence, it acts as a controller reading and writing to the peripheral devices over the system interconnect bus. At the same time, it also supports commands triggering peripheral-specific actions through single-wire event lines as simpler peripheral-event handling systems do (Section II). These are referred to as *instant actions*. An overview of PELS's operation modes and key differences compared to SOTA is provided in Figure 1.

*1) **Link**:* To provide parallelism when servicing multiple peripheral linking events, PELS is internally organized into independent linking units, referred to as *links*, as shown in Figure 1c. When each link executes a sequenced action, the interconnect topology determines the number of parallel interactions with the peripheral devices, while its arbitration policy affects each link's typical and maximum latency, especially in the worst-case scenario where all links try to access peripherals simultaneously. As shown in Figure 2, a link consists of three main building blocks: the *trigger unit*, the *execution unit*, and the *SCM*, described in the following.

*a) **Trigger unit**:* Input events to PELS are broadcasted to all links. Each link's trigger unit masks incoming events ❶, further checked against trigger conditions ❷, such as *all-selected-active* (`AND`) or *any-selected-active* (`OR`). For instance, a trigger condition can be a threshold to generate an event. The main CPU configures both masking and triggering conditions through each link's private configuration registers ❸.

*b) **Private SCM**:* If the condition to propagate the event is satisfied, a trigger signal is generated and buffered with a FIFO to prevent interference with a running execution unit. A finite state machine (FSM) reads line-by-line the microcode to be executed from the integrated instruction memory ❹. Each line in the memory holds one *command*. Dedicated instruction memory allows for predictable latency due to not having to contend on a shared bus, which can be as low as a single clock cycle when linking with one interacting peripheral. The instruction memory is implemented with SCMs, which incurs minimal power consumption and area overhead for small memory footprints, as in the case of PELS. With a SRAMs, the sense amplifiers would otherwise dominate the area [20].

*c) **Execution unit**:* Figure 2 highlights the microarchitecture of the execution unit. One clock cycle after a successful triggering condition, the execution unit receives the first command from the instruction memory ❺. In the simplest case, this command triggers an instant action through single-bit event lines directly connected to the peripheral devices.

Otherwise, PELS can issue sequenced actions for peripherals without dedicated event lines. When the command becomes available, the execution unit issues a read operation to the system interconnect, and it stalls until it receives the register's value from the peripheral ❻. The value is then modified ❼ according to the command. One cycle after the read succeeds,

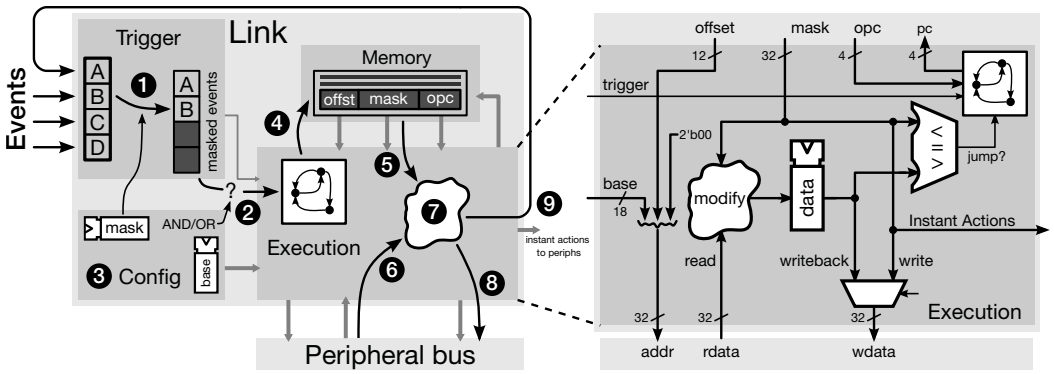


Fig. 2. Architectural overview of a single link. The figure highlights the execution unit microarchitecture.

the modified value is written back ❽ to the peripheral register. Finally, Figure 2 shows that some event lines triggered by instant actions can be looped back to the incoming events, which enables inter-link triggering ❾.

*2) **Commands**:* The primitive `write` command writes a known value to a register. More complex and higher latency commands read the value from a selected register, perform a bitwise operation, and write the result back to the register (*read-modify-write*). They comprise `set`, `clear`, and `toggle` commands. Some use cases for such commands include setting/resetting peripheral register flags or toggling GPIOs.

Read-modify-write commands constrain the microcode encoding, as for every peripheral register access, an address and a mask are needed. Provided that a memory transaction for bus-based peripheral interaction requires both the datum and an address, a single-cycle transaction needs more than 32-bit encoding. To reduce the address encoding space, PELS only requires a word-addressed offset relative to a base address specific to each link. Overall, the number of offset bits lies within a range of 10-14 bits, leading to a command encoding with 4-bit *op-code*, a 12-bit field address, and a 32-bit data.

To address the use-case of sensor readout followed by a threshold comparison (a common operation for event-linking, see Figure 3), we include a `capture` and `jump-if` command. `capture` performs a masked read and stores it in a single 32-bit register per link. `jump-if` reads the value from the datapath register and compares it against its 32-bit operand. If the condition is satisfied, it jumps to another command.

Two additional commands, namely `loop` and `wait`, are implemented to enable the design of non-nestable hardware loops and wait counters. In particular, they subsume watchdog-like functions without requiring an external timer.

Finally, the `action` command is the core of the *instant action* mechanism. It sets or toggles one or multiple outgoing single-wire event lines. The 12-bit field reserved for addresses with sequenced actions is instead used to select a group of event lines, while the 32-bit operand is used to set them. The single-wire event lines enable straightforward integration of any peripherals supporting built-in actions as SOTA approaches shown in Figure 1b. Additionally, PELS allows links to trigger each other through specific instant actions. This feature is beneficial when links have limited instruction memory or can access specific peripherals only, and enables links specialization and diversification in handling a subset of peripherals, a feature missing in existing SOTA.

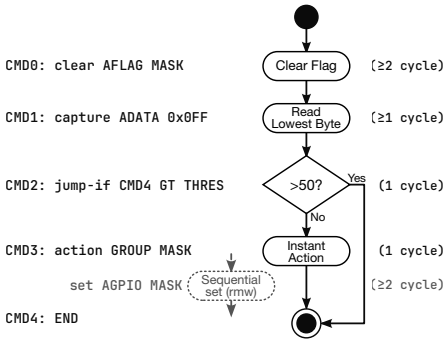


Fig. 3. Pseudocode showing PELS's flexibility in providing both sequenced and instant actions and associated latency in clock cycles.

*3) **Programming model**:* Figure 3 provides an example of PELS's dual-mode operation flexibility. In the presented pseudocode, PELS implements a threshold-triggered operation following sensor readout (e.g., a thermistor/varistor) with both instant and sequenced actions. In the former mode, the output event line is redirected to the peripheral (e.g., a GPIO) if the latter is co-designed to support single-wire event lines. With sequenced actions instead, PELS directly controls the peripheral through the peripheral interconnect, regardless of its event interface design. The figure shows the latency of each stage in clock cycles; as discussed in Section III-1, instant actions have lower, fixed latency but require specialized logic.

IV. Evaluation

In this section, we first introduce the RISC-V MCU where PELS is integrated (Section IV-A). Then, we evaluate PELS's beneficial impact on the MCU power consumption when mediating peripheral linking operations (Section IV-B) and low silicon area overhead (Section IV-C), as a result of the design tradeoffs described in the previous section.

A. PULPissimo RISC-V MCU

PULPissimo [12] is a single-core, MCU-class RISC-V system designed for embedded ULP applications. Among the features it supports to achieve ULP while retaining high en-

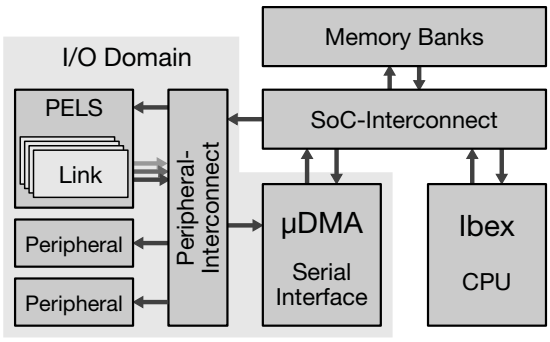


Fig. 4. PELS integration in PULPissimo SoC.

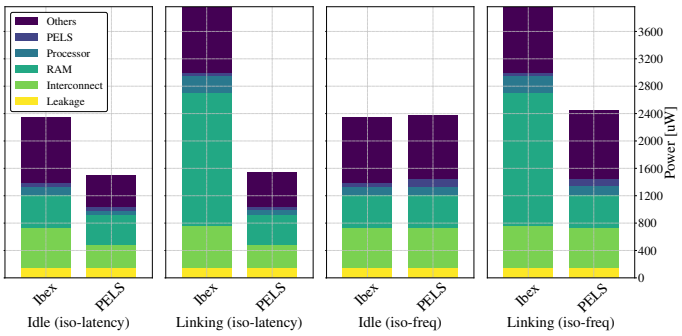


Fig. 5. Power estimation with iso-latency and iso-frequency conditions between PELS and Ibex.

ergy efficiency, we mention: (i) an autonomous I/O DMA to decouple data collection and processing, (ii) near-threshold operation [21], and (iii) adaptive body-biasing to reduce standby leakage power [9]. We opt for the single issue, in order, 2-pipeline stage Ibex core [22] among the processors supported by the MCU because of its minimal area (Section IV-C).

In this work, we extend PULPissimo with PELS as depicted in Figure 4. The main configuration parameters to tune PELS performance within a system are the number of links (number of parallel actions) and the memory size (number of commands per link). Furthermore, when issuing *sequenced actions*, the topology of the system interconnect and its arbitration policy affect (i) the number of links that can access a group of peripherals in parallel and (ii) the system's predictability due to variable latency. In the present work, we rely on PULPissimo's implementation of round-robin arbiters in the system interconnect to guarantee fair bandwidth distribution.

B. Functional performance

To evaluate the architecture's performance, we design an event-linking application consisting of a threshold-crossing check after I/O DMA-managed sensor readout through the SPI interface, similarly to the example shown in Figure 3. We compare PELS's mediation through sequenced actions with an interrupt-based mechanism redirecting the linking event to the Ibex core in two scenarios: (i) iso-latency, i.e., PELS and Ibex are independently clocked at different frequencies to meet specific latency requirements and (ii) iso-frequency without latency constraints. Both evaluations on such benchmarks are carried out with cycle-accurate RTL simulations, and we further estimate the power consumption of a linking event using Synopsys PrimeTime on the synthesized netlist. We present power estimations for sequenced actions, as instant actions introduce negligible dynamic power.

Figure 5 provides a breakdown of PULPissimo power consumption when waiting for and handling the event linking for both iso-latency and iso-frequency scenarios. In the iso-latency scenario, PELS and Ibex match a $500\,\mathrm{ns}$ latency requirement at $27\,\mathrm{MHz}$ and $55\,\mathrm{MHz}$ respectively. The power consumed during the event linking process in the SoC is reduced by $2.5\times$ when handled by PELS compared to the Ibex interrupt-driven mechanism. We argue this is due to the decreased frequency (affecting the dynamic power) to meet the imposed latency constraint and the reduced switching activity around the memory system ($3.7\times$ less power). Analogously, waiting for an event (*idle* mode) without standby leakage power-saving techniques uses $1.5\times$ less power with PELS-driven event linking.

In the iso-frequency case, both systems are clocked at $55\,\mathrm{MHz}$. This scenario also shows the benefits of the event-linking system in the active mode, with a power consumption reduction of $1.6\times$. As in the previous case, one of the main contributions to the reduced power envelope comes from the little activity around the memory system in the PELS-driven scenario ($4.3\times$ less power draw).

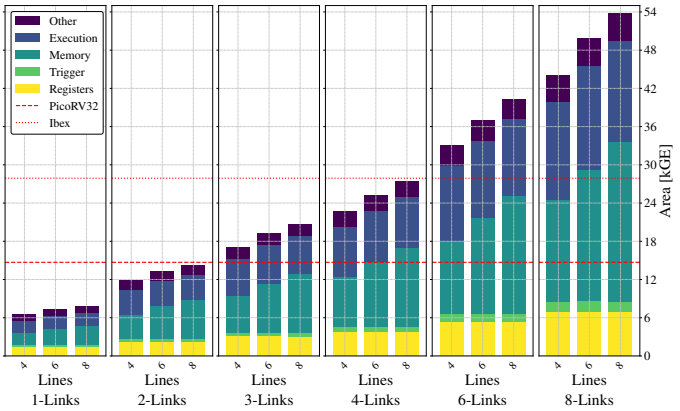
For a latency comparison, we observe 7 and 2 clock cycles for sequenced and instant actions, respectively. The former is slower and depends on the peripheral bus protocol (APB for [12]), while the latter is fixed. As anticipated, Ibex takes significantly longer (16 clock cycles) to handle the same linking event because of the interrupt handling mechanism overhead.

C. Implementation

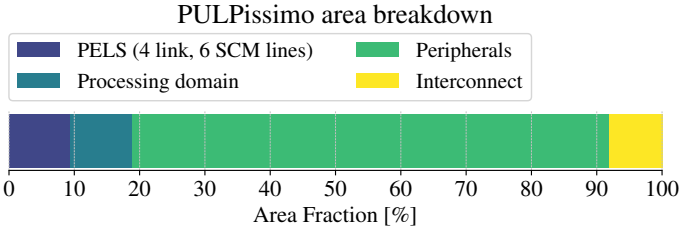
We synthesize PELS in several configurations and compare it against extremely low-footprint processors that could serve as fully-fledged general-purpose solutions for event-linking, proving the benefits of the proposed microcode approach. Topological synthesis is carried on with Synopsys Design Compiler 2022.03, targeting TSMC's $65\,\mathrm{nm}$ technology at $250\,\mathrm{MHz}$, TT corner, and $25\,^{\circ}\mathrm{C}$. We vary the number of links from 1 to 8, sufficient to cover realistic use cases. For each link, we consider SCMs with 4, 6, and 8 lines (microcode commands). Figure 6a shows the results. PELS compares favorably with some of the leanest general-purpose cores in the RISC-V landscape, represented as dashed horizontal lines in the figure, that we synthesize with equal frequency and Process, Voltage, Temperature (PVT) conditions for a fair comparison. In its minimal configuration (1 link, 4 commands), PELS results in an area of about $7\,\mathrm{kGE}$, $4\times$ smaller than Ibex [2] (about $27\,\mathrm{kGE}$) and $2\times$ smaller than PicoRV32 [3] (about $14.5\,\mathrm{kGE}$), without considering the external SRAMs that those RISC architectures require for load and store operations.

PELS proves to be lightweight also within the RISC-V SoC it is integrated into. Figure 6b shows that a *4-links* PELS accounts for about 9.5% PULPissimo's area (1% when considering the SRAM - $192\,\mathrm{KiB}$ in the implemented configuration).

[2]https://github.com/lowRISC/ibex
[3]https://github.com/YosysHQ/picorv32



(a)



(b)

Fig. 6. PELS silicon area evaluation in TSMC's 65 nm technology, 250 MHz, TT corner, 25 °C. **(a)** PELS area sweep over the number of links and SCM lines. **(b)** PELS area overhead in PULPissimo.

The presented evaluation shows that PELS is highly competitive in terms of silicon area, power consumption, and latency compared to both general-purpose solutions and existing SOTA addressing event-linking with bespoke event interconnects. Instead, PELS resolves these limitations by combining *instant* and *sequenced* actions with an SCM-based memory system, guaranteeing flexibility and fast access to private memory.

V. Conclusion

In this paper, we address the challenge of designing flexible near-sensor processing systems that can perform fully autonomous multi-sensor and inter-peripheral communication. To achieve this goal, we propose a peripheral-agnostic, fully autonomous, lightweight, flexible Peripheral Event Linking System (PELS) with microcode programmable event-linking. PELS supports both single-wire event lines (instant actions) to optimize for stringent low-latency requirements and traditional commands issued through the system interconnect bus targeting peripheral configuration registers (sequenced actions). We integrate it in an ULP RISC-V IoT processor and show that the power consumption of a linking event is reduced by 2.5 times compared to approaches that rely on the main core for the event-linking process at a low silicon area of 7 kGE in its minimal configuration when instantiated in a ULP RISC-V IoT processor.

Acknowledgments

This work is supported by the dAIEDGE (101120726) and TRISTAN (101095947) projects, funded by HORIZON-CL4-2022-HUMAN-02 and HORIZON-CHIPS-JU, respectively.

References

[1] S. Lu, J. Lu, K. An, X. Wang, and Q. He, "Edge computing on iot for machine signal processing and fault diagnosis: A review," *IEEE Internet of Things Journal*, pp. 1–1, 2023.

[2] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "14.5 envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fd-soi," in *IEEE ISSCC*, 2017, pp. 246–247.

[3] R. M. Abdelmoneem, E. Shaaban, and A. Benslimane, "A Survey on Multi-Sensor Fusion Techniques in IoT for Healthcare," in *2018 13th ICCES*, 2018, pp. 157–162.

[4] Y. He, J. Guo, and X. Zheng, "From surveillance to digital twin: Challenges and recent advances of signal processing for industrial internet of things," *IEEE S. P. Magazine*, vol. 35, no. 5, pp. 120–129, 2018.

[5] K. Shimada, K. Sano, K. Fukuoka, H. Morita, M. Daito, T. Kamei, H. Hamasaki, and Y. Shimazaki, "A 33kdmips 6.4w vehicle communication gateway processor achieving 10gbps/w network routing, 40ms can bus start-up and 1.4mw standby power," in *ISSCC*, 2023, pp. 240–242.

[6] P. Saha and S. Mukhopadhyay, "Multispectral Information Fusion With Reinforcement Learning for Object Tracking in IoT Edge Devices," *IEEE Sensors Journal*, vol. 20, no. 8, pp. 4333–4344, 2020.

[7] T. He, L. Zhang, F. Kong, and A. Salekin, "Exploring inherent sensor redundancy for automotive anomaly detection," in *DAC*, 2020, pp. 1–6.

[8] J. Steinbaeck, C. Steger, G. Holweg, and N. Druml, "Next generation radar sensors in automotive sensor fusion systems," in *SDF*, 2017.

[9] D. Rossi, I. Loi, A. Pullini, C. Müller, A. Burg, F. Conti, L. Benini, and P. Flatresse, "A self-aware architecture for pvt compensation and power nap in near threshold processors," *IEEE Design & Test*, vol. 34, no. 6, pp. 46–53, 2017.

[10] D. Rossi, F. Conti, M. Eggiman, A. D. Mauro, G. Tagliavini, S. Mach, M. Guermandi, A. Pullini, I. Loi, J. Chen, E. Flamand, and L. Benini, "Vega: A ten-core soc for iot endnodes with dnn acceleration and cognitive wake-up from mram-based state-retentive sleep mode," *IEEE JSSC*, vol. 57, no. 1, pp. 127–139, 2022.

[11] A. Pullini, D. Rossi, G. Haugou, and L. Benini, "μDMA: An autonomous I/O subsystem for IoT end-nodes," in *2017 27th PATMOS*, 2017, pp. 1–8.

[12] P. D. Schiavone, D. Rossi, A. Pullini, A. Di Mauro, F. Conti, and L. Benini, "Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX," in *2018 IEEE S3S*, 2018, pp. 1–3.

[13] *AN0025: Peripheral Reflex System*, Silicon Labs. [Online]. Available: https://www.silabs.com/documents/public/application-notes/an0025-efm32-prs.pdf

[14] "New functions of the rl78/g23 logic and event link controller (lelc)," Tech. Rep. [Online]. Available: https://www.renesas.com/us/en/document/whp/new-functions-rl78g23-logic-and-event-link-controller-elcl?language=en

[15] *Getting Started with Core Independent Peripherals on AVR*, Microchip. [Online]. Available: https://ww1.microchip.com/downloads/en/Appnotes/Microchip%20Getting%20Started%20with%20Core%20Independent%20Peripherals%20DS00002451B.pdf

[16] *PPI - Programmable Peripheral Interconnect*, Infineon. [Online]. Available: https://infocenter.nordicsemi.com/topic/ps_nrf52810/ppi.html

[17] *AN4640 Application note: Peripherals interconnections on STM32F4x series*, STMicroelectronics. [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/application_note/59/ed/30/07/55/76/4e/82/DM00154959.pdf/files/DM00154959.pdf/jcr:content/translations/en.DM00154959.pdf

[18] "Tutorial: Introducing the xgate module to consumer and industrial application developers," Tech. Rep. [Online]. Available: https://www.nxp.com.cn/docs/en/application-note/AN3224.pdf

[19] R. A. Bjørnerud, M. W. Lund, and K. Svarstad, "Event control and programming for microprocessor peripheral systems," in *NORCHIP*, 2009, pp. 1–4.

[20] A. Teman, D. Rossi, P. Meinerzhagen, L. Benini, and A. Burg, "Power, area, and performance optimization of standard cell memory arrays through controlled placement," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 21, no. 4, may 2016. [Online]. Available: https://doi.org/10.1145/2890498

[21] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices," *IEEE TVLSI*, vol. 25, no. 10, pp. 2700–2713, 2017.

[22] lowRISC, *lowRISC Ibex Reference Guide*, lowRISC. [Online]. Available: https://ibex-core.readthedocs.io/en/latest/03_reference/index.html