



Article

From SW Timing Analysis and Safety Logging to HW Implementation: A Possible Solution with an Integrated and Low-Power Logger Approach

Francesco Cosimi ^{1,2,*} , Antonio Arena ², Paolo Gai ²  and Sergio Saponara ¹

¹ Department of Information Engineering, University of Pisa, 56122 Pisa, Italy; sergio.saponara@unipi.it

² Huawei Pisa Research Center, 56121 Pisa, Italy; antonio.arena1@huawei.com (A.A.); paolo.gai@huawei.com (P.G.)

* Correspondence: francesco.cosimi@huawei.com

Abstract: In this manuscript, we propose a configurable hardware device in order to build a coherent data log unit. We address the need for analyzing mixed-criticality systems, thus guaranteeing the best performances without introducing additional sources of interference. Log data are essential to inspect the behavior of running applications when safety analyses or worst-case execution time measurements are performed. Furthermore, performance and timing investigations are useful for solving scheduling issues to balance resource budgets and investigate misbehavior and failure causes. We additionally present a performance evaluation and log capabilities by means of simulations on a RISC-V use case. The simulations highlight that such a data log unit can trace the execution from a single- to an octa-core microcontroller. Such an analysis allows a silicon developer to obtain the right sizings and timings of devices during the development phase. Finally, we present an analysis of a real RISC-V implementation for a Xilinx UltraScale+ FPGA, which was obtained with Vivado 2018. The results show that our data log unit implementation does not introduce a significant area overhead if compared to the RISC-V core targeted for tests, and that the timing constraints are not violated.

Keywords: automotive; data log; execution tracing; hardware; RISC-V; safety; ISO 26262; timing analysis



Citation: Cosimi, F.; Arena, A.; Gai, P.; Saponara, S. From SW Timing Analysis and Safety Logging to HW Implementation: A Possible Solution with an Integrated and Low-Power Logger Approach. *J. Low Power Electron. Appl.* **2023**, *13*, 59. <https://doi.org/10.3390/jlpea13040059>

Received: 1 August 2023

Revised: 12 September 2023

Accepted: 26 September 2023

Published: 2 November 2023



Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Many modern vehicles contain large quantities of electronic systems that cooperate and communicate together. With the increasing number of such systems, the complexity of running these applications exponentially increases, requiring time-consuming safety analyses in order to identify potential failures and malfunctioning risks. The ISO 26262 (2018) standard standardizes the safe usage of electronic devices and defines functional safety (FuSa) as “the absence of unreasonable risk due to hazards caused by malfunctioning behavior of electrical or electronic systems” [1]. Risk classification comes with four automotive safety integrity levels (ASILs), from ASIL A (the lowest level of risk) to ASIL D (the highest level of risk) [2]. ISO 26262 requires applications and electronic devices to match the right level of risk based on the criticality of the running application.

Moreover, automotive systems have to ensure the coexistence of multiple applications that may have different ASILs, may run in the same hardware platform, and may share the same resources, as shown in Figure 1. We commonly refer to these scenarios as mixed-criticality systems (MCSs). Software (SW) techniques combined with hardware (HW) virtualization extensions provide a support framework to guarantee both the worst-case execution time (WCET) and the freedom from interference (FFI) standards for applications running in MCSs.

ISO 26262 defines FFI as the “absence of cascading failures between two or more elements that could lead to the violation of a safety requirement”, where “element” refers to a “system or part of a system including components, hardware, software, hardware

parts, and software units". Meanwhile, a "cascading failure" is a "failure of an element of an item causing another element or elements of the same item to fail".

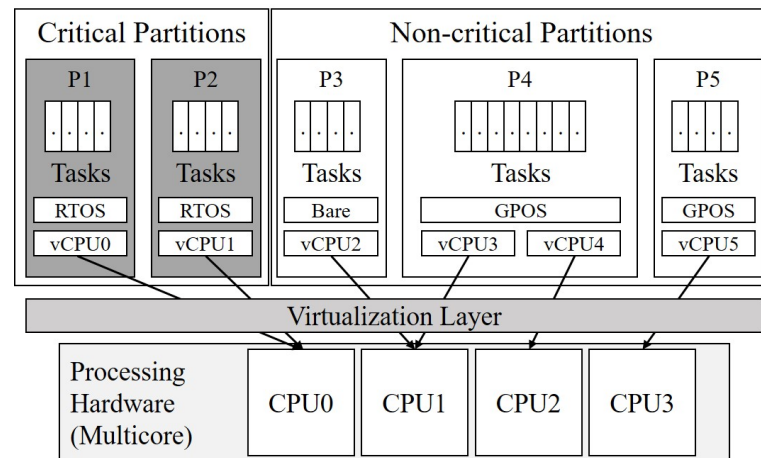


Figure 1. Virtualization example for a multi-core MCU. Here, we have critical partitions managed by a real-time OS, and non-critical ones managed by a general-purpose OS (or bare metal). The virtualization layer can also allocate more virtual CPU cores (vCPU) than the number of physical CPU cores.

Furthermore, the advent of multi-core and many-core processors, which typically provide massive virtualization support and resource sharing, has reduced the significance of analysis for end-to-end execution with respect to timing validation and verification (V&V); in addition, it has affected FFI and WCET matching. Applications running on different cores might interfere with each other, thereby leading to instruction latency increases and losses in the FFI in the system. This scenario typically jeopardizes critical task executions, thus introducing an unpredictable factor and a scenario complexity increase, making the estimation of WCET increasingly difficult [3]. For this reason, timing analysis methods have become crucial for the correct development of automotive MCSs since the execution awareness and knowledge of hardware health states are essential to correctly balance resources.

This manuscript presents a design, sizing, simulation, and implementation of a hardware log mechanism called the data log unit (DLU), which can be easily embedded in integrated circuits. The DLU is capable of collecting data from different data sources and building a single and coherent data output to be further analyzed. In particular, in this project, we focus on collecting the data from customized peripherals that perform execution tracing, performance evaluation, and error management.

The paper is organized as follows: Section 2 compares our data log unit with state-of-the-art technology, as well as highlights the key points of our upgrades. Section 3 describes the timing and performance analysis in critical systems, and it introduces RISC-V technology and the gaps with other technologies that have been detected by the community. Finally, we present the function that the DLU has in the scope of our project. Section 4 explains the contribution the DLU provides to the timing and performance analysis, as well as the system validation. Section 5 explains the architecture of the DLU with a description of its timing and HW behavior. Moreover, Section 6 shows the results of the behavioral simulations of the log mechanism, thereby analyzing different configurations and settings to obtain the right sizings of the devices. Section 7 reports the synthesis and implementation of the log devices (through Vivado 2018), and discusses the cost of our design in terms of area and power overhead for a Xilinx UltraScale+ FPGA. Finally, Section 8 concludes the manuscript.

2. Related Works

The DLU is a device that is capable of collecting and transmitting data from several heterogeneous sources. In the literature, it is possible to find certain references about similar implemented hardware. For example, Table 1 presents a list of patents that target related issues, but refer to different applications and contexts. In [4–7], the design choices for implementing a logging mechanism heavily depend on the input sources considered, such as memory usage, data from sensors, or the status of other peripherals. In this manuscript, we propose the possibility of customizing the input sources. Moreover, we defined a standard interface between the data sources and the logger itself in order to reduce the DLU complexity. The major novelty introduced with our DLU is the possibility to adapt the hardware in an easy way to different configurations of the input. The DLU is designed to interact with multiple different sources and generate a unique data log that can be easily analyzed. Furthermore, the DLU does not introduce any software overhead for performing logging routines since the data are entirely collected through the hardware.

Table 1. List of patents similar to the DLU.

Patent N°	Title
US5944841A [4]	Microprocessor with built-in instruction tracing capability
RU2639013C1 [5]	Device for recording and transmitting the data of movable property objects
US9912531B2 [6]	Data logging or stimulation in automotive Ethernet networks when using vehicle infrastructure
CN104520674B [7]	Movable property data logger and transmitter

Moreover, the literature provides examples of log mechanisms designed for embedded systems, specifically aimed at collecting data about the execution and behavior of the system being monitored. For example, in [8], the authors propose an FPGA implementation of a modular data logger, where the modules can have various functions. There are modules that aim to connect to devices, like an ADC (analog-to-digital converter), modules to store the data (e.g., on an SD card), and other modules to process the data and control the data fetching, like filters or timers. Additionally, the modular design makes it particularly easy to add new devices, data sinks, or any other function. The main difference between [8] and our work is that we do not exclusively work with Linux-based systems. We can collect data from different sources, and the DLU and its proxies are meant to be compatible with any architectural and software environments.

In [9], the authors investigated the design of the embedded data acquisition systems by focusing on the mobility, efficiency, and coverage of as many use cases as possible. They studied the feasibility of a universal mobile data acquisition system with a proof-of-concept design and implementation that covered multiple solutions. Their goal was to create a basic framework for universal data acquisition systems that are capable of handling digital sensors. The data logger device is the core component of the data acquisition system as it captures data from the sensor interfaces and either stores it locally or transmits it to a network server. The key difference with our design is that the work presented in [9] was meant to collect, store, and transmit data from digital sensors and actuators. Meanwhile, our DLU, if properly configured and connected, is capable of logging the data regarding activated tasks, the number of hardware events, and the amount and type of error management actions taken.

We can also affirm the same for [10], where the data from sensors and actuators regarding an LED street lighting system were collected. However, we do not have any info about the real task execution and system hardware behavior of that model, but such information is available through the DLU.

The data acquisition system proposed in [11] differs from our design in many aspects. In [11], the FPGA log system's main target involves the acquisition of data from analog-to-digital converter devices, and this constitutes a huge limit with regard to the flexibility of inputs. Moreover, the designed FPGA circuit was composed of only an N-bit counter and a buffer filled with incoming data. Our design, instead, allows one to manage the data from different data sources and control the inputs at runtime through a software application.

Finally, Ref. [12] proposes a design that allows for a flexible logging operation in terms of module and interface responsibility separation. The embedded controller (HW+SW) is the device responsible for data logging and data processing between the monitored embedded device and the microcontroller. An embedded controller is associated with every data source, and it has a consistent increase in costs in terms of area and power consumption. Meanwhile, in our design, we propose a hardware-only master device, which cooperates with some slave devices and produces a single log from multiple data sources. This reduces the costs of the application, and it minimizes the interference between the running software and the log routines.

3. The DLU Background

This section introduces the background of the hardware log mechanism. In particular, we first define the importance of the timing and performance analysis. We then describe the possible open-source RISC-V solutions. Finally, we focus on the role that the DLU has in the context of our project.

3.1. Timing and Performance Analysis Background

Timing analysis tools use performance data, software-tracking results, and log of error recovery actions for an offline examination of execution. These tools are useful for highlighting timing issues and their causes, as well as mitigating the interference at the development phase or in a post-mortem analysis. Currently, the market offers many solutions in terms of performance and execution monitoring, as described in [13]. Software tools (e.g., Eclipse TraceCompass [14], SEGGER SystemView [15], Percepio TraceAlyzer [16], Rapita RapiTask [17], Gliwa T1 [18], and Lauterbach TRACE32 [19]) can cooperate with the automotive application to obtain the needed results. However, the usage of software solutions typically generates interference with execution by introducing a source of unpredictability into the timing validation process. Moreover, some of the existing solutions may also need external hardware support to perform a complete execution tracing.

3.2. RISC-V Background

RISC-V is a free and open instruction set architecture (ISA) that enables a new processor innovation era through an open standard collaboration. The RISC-V ISA delivers a new level of free, extensible software and hardware freedom on architecture, and it combines a modular technical approach with an open-source license business model. Moreover, open-source licensing makes it the right choice for researching and prototyping [20]. RISC-V offers many different solutions for processing units. Ready-to-implement HDL cores for FPGA (Verilog or VHDL) are available, and there is the possibility of adapting them to our requirements.

Certain notable architectural examples that are easily accessible for prototyping are the 32-bit RI5CY microcontroller core and the 64-bit Ariane microprocessor core [21]. Furthermore, the online community offers multi-core architectures, such as OpenPULP HERO (RI5CY), and OpenPiton (Ariane) [22,23].

We investigated the solutions proposed by the RISC-V development community. In particular, we focused on the works carried out by the Special Interest Group Safety (SIG-Safety) [24]. The goal of SIG-Safety is to identify the gaps in various RISC-V specifications and their implementations, in order to provide guidelines on how to realize RISC-V-based products that are suitable for safety-critical applications. In fact, RISC-V standard technologies have not yet implemented ASIL requirements to obtain a complete automotive implementation.

The following list presents a set of hardware safety features that were designed to satisfy ASIL processor requirements identified by the RISC-V SIG-Safety group, which can be integrated into a RISC-V environment:

- Safety monitors for hardware performance (such as performance counters or a performance monitoring unit) to detect hardware misbehavior during runtime and facilitate timing analyses.
- Error-correcting code (ECC) designed to detect and rectify data bit and address errors.
- Hardware stack protection to check the overflow and underflow of the designated stack space, coupled with code protection to prevent unauthorized reading and writing to the code space.
- Programmable watchdog timer to detect and recover from a deadlock situation.
- Lock-step interface and redundant processors to create a redundant (in-time) execution.
- Memory protection unit (MPU) to define regions and assign access attributes, ensuring varying protection levels.
- Quality of service (QoS) management and autonomous recovery action management.
- Embedded execution tracing and logging capabilities.

3.3. Project Development Background

The DLU is part of a research project that includes a set of core independent peripherals and a proper software library implementation. After the FuSa requirements were implemented and the RISC-V capabilities analysis was conducted, we identified the devices listed below (shown in Figure 2) as suitable for our needs, and meaningful in terms of functional safety improvement:

- A performance monitoring unit (PMU), as described in [25], collects hardware execution data and provides results to the software. The events collected can influence FuSa or performance improvement, and their statistical importance in execution has to be considered [26].
- The error management unit (EMU) receives information about SW/HW errors or misbehavior. Then, a reaction can be applied, either with a quality of service (QoS) degradation or a fail-safe strategy. In MCSs, if a minimum QoS is guaranteed for critical applications, the system takes actions to ensure, at the very least, the execution of a degraded functionality.
- An execution tracing unit (ETU), as described in [27], is essential for tracing software events. This unit is designed to be AUTOSAR run-time interface (ARTI) compliant [28,29]. ARTI defines the standard of communication between the AUTOSAR applications and timing analysis tools for the system inspection.
- The time management unit (TMU) is a service peripheral. It provides a coherent time source (global peripheral time, GPT) to the peripherals.

In this manuscript, we focus on PULPissimo [21], a platform with a single RI5CY 4-stage 32-bit core for Xilinx UltraScale+ FPGA, as the use case for the implementation and testing of our devices.

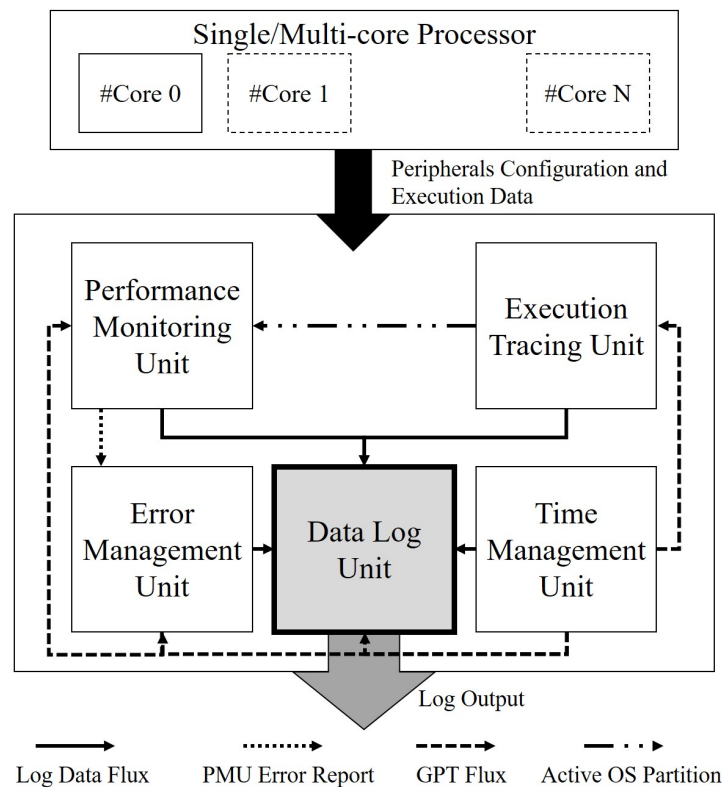


Figure 2. Block scheme of the safety peripherals' implementation. The DLU is highlighted in gray. The figure shows how the DLU receives data from all of the other peripherals in order to build a coherent and unique log.

4. Paper Contribution

In this manuscript, we present hardware support for the data log unit for the purpose of efficient data collection, as shown in Figure 2. This is useful for performance evaluation, resource usage estimation, resource budgeting, execution flow analysis, error management, and recovery tracking.

The goal of the DLU is to reduce the interference between the running applications and the tracing mechanisms while providing an instrument to record concurrent executions on the same hardware platform. Moreover, we integrated the DLU with a specific set of peripherals to obtain all the functionalities mentioned above. Through the processing of device output data, we can perform an accurate timing and performance analysis; in addition, we can investigate the interference between the applications with different ASILs, and prevent future possible misbehavior.

The DLU collects data in a standardized way from all of the sources that are connected, and it also builds a complete log. Information passes through a system of buffers and streams out through the output. The interface with the outer world can be I2C, SPI, Ethernet, etc., according to the user's needs [30]. We plan to implement this communication port with custom HW devices without requiring any SW actions in order to limit the overhead on the main CPU. The stream of data from the output of the DLU will be collected by a proper architecture that is developed outside the scope of this project.

5. DLU Design and Features

5.1. DLU Design

The block schemes presented in Figures 3 and 4 show the device connected to N different data sources. By design, the maximum number of sources (and ports) running at the same time in our implementation is equal to 32. Every input port has a hard-wired identifier that allows us to understand, in a unique way, what the data source is. The DLU

cyclically checks such input ports to verify if there are some available data to be transmitted. The capability of the DLU to manage up to 32 inputs strictly depends on the frequency and amount of new data to be logged.

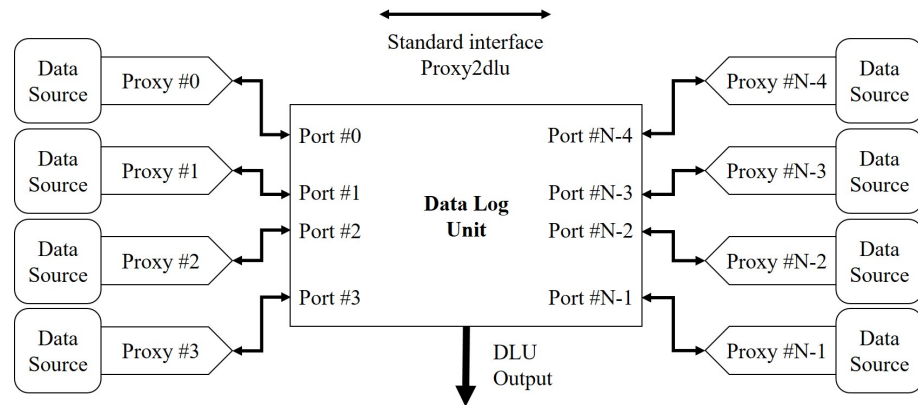


Figure 3. DLU scheme of the external connections with data sources. In our case, the input data came from custom-designed safety peripherals. Each proxy was connected with the corresponding data source and port. The DLU manages the inputs and provides a unique output through the DLU output port.

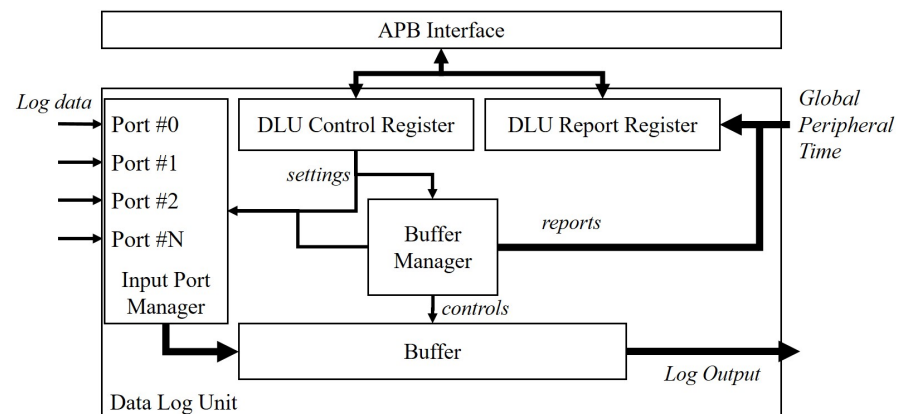


Figure 4. DLU scheme of the internal device structure. The buffer manager reads the DLU control register settings in order to control the input port manager and the buffer itself. The buffer is dumped through the log output, which is connected to the DLU output. The control and report register are accessed by the software through an advanced peripheral bus (APB) interface.

The buffer manager circuit plays the role of controller for the buffer status. This is performed for the purpose of scheduling the active input ports, as well as for the status of the proxy buffers (which will be described later in this section). The settings for the input port manager and buffer manager can be found in the control register. If an overflow (of the proxy's or DLU's buffer) is detected in any buffer, the DLU records the information and makes it available in the log or at the runtime through the DLU report register. The buffer is divided into two sections. Every time a DLU's buffer section is filled, it is then ready to be dumped. After this, a *data_ready* signal rises. The *data_ready* signal triggers the activation of the interface with the outer world. The interface can be I2C, SPI, or Ethernet, depending on the system's configuration. The implementation of a communication port with custom HW devices, without any SW intervention, limits the overhead on the main CPU and frees the application's software from any interference due to the log routines themselves.

5.2. Proxy Design

Figure 5 presents the internal architecture of the DLU proxy. The proxy is needed to standardize the interface between the DLU and the input sources, as well as manage the input traffic through a FIFO mechanism. The device contains a FIFO buffer, which is divided into sections and subsections, a manager responsible for directing packets and checking their status, and a mux–demux system to manage the input and output paths. Each section represents the portion of the log transmitted to the DLU when requested. The following equation determines the size of a proxy buffer:

$$Size(Byte) = 4 \cdot N_Sec \cdot N_SubSec \cdot Report_Reg \quad (1)$$

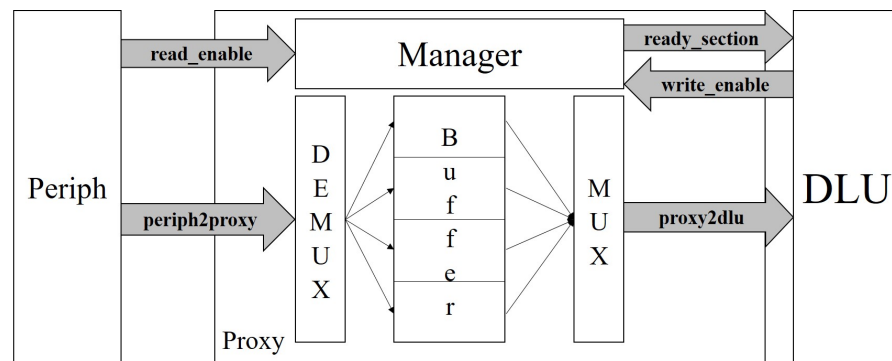


Figure 5. Block scheme of a proxy. This picture shows how the interface between the peripheral and the DLU is. The manager transfers data to and from the FIFO buffer.

Specifically, N_Sec is the number of sections in the buffer, N_SubSec are the subsections in each section, and $Report_Reg$ is the size (in 32-bit words) of the peripheral’s report register that corresponds to the number of registers copied at the same time in a single subsection. This particular sizing of the proxy’s buffer avoids data corruption during the log.

In this manuscript, we consider the $Report_Reg$ size fixed for each peripheral, and we analyze how to tune the number of sections and subsections in the following chapters. Such parameters are essential for the correct behavior of the system; also, they need to be finely tuned to avoid data overproduction, excessive area usage, and power consumption overhead.

5.3. Data Source–Proxy and Proxy–DLU Interfaces

In this section, the data source–proxy and the proxy–DLU interfaces are presented. The data source–proxy—in a single clock cycle—takes as input all the info to be logged, and this is generated by the peripheral connected to it. Figure 6 shows the signal sequences required to transfer data from a peripheral to the buffer inside the data source–proxy interface. Once the device contains the available data, it raises a *read_enable* signal and loads data onto the *periph2proxy* BUS, such that the data can be copied into the proxy’s buffer within a single clock period. Moreover, the proxies directly copy data from the implemented peripheral registers into their buffers, thus reporting on the information about the execution.

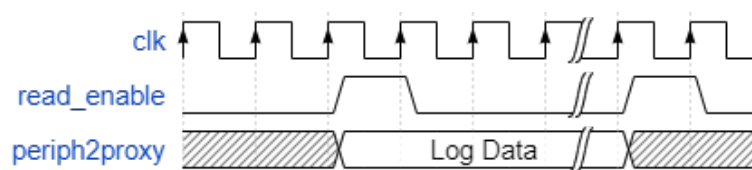


Figure 6. Representation of the timing sequence of the communication signals between the safety peripheral and the proxy. Transmission starts every time the *read_enable* signal reaches the proxy.

The size of the *periph2proxy* BUS matches that of the words that make up the entire information set to be logged. The data are copied in a portion of the buffer, and the

manager waits until an entire section is filled. The size of a subsection is a multiple of the number of registers transferred from the peripheral to the log; this is performed to avoid data corruption.

Figure 7 shows the timing sequence of the proxy–DLU port data transfer. When a section inside the proxy’s buffer is full, the proxy raises the *ready_section* signal, and it then waits for the right to transmit. Once the DLU allows the proxy to transmit the data, it raises the *write_enable* signal for the corresponding port. The signal triggers the transmission of the data through the 32-bit BUS *proxy2dlu*.

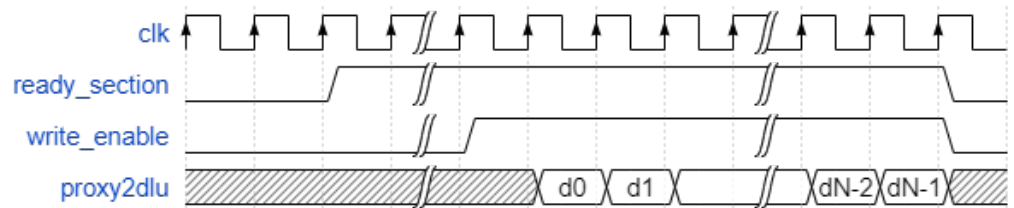


Figure 7. Representation of the timing sequences of the communication signals between the proxy and the DLU’s port. If a section is ready, then the *ready_section* signal is raised and the proxy waits for the DLU to raise the *write_enable* signal to begin transmission to the DLU’s port.

6. DLU Sizing within a Python-Based Environment

In order to test the functionalities of our DLU and to create a logging mechanism for profiling, we implemented a Python script that simulates the behavior of the data source devices. The main purposes of such a simulation are to evaluate the DLU performance, estimate the correct buffer sizing, and avoid under-/over-sizing, such that we can reduce costs and minimize data loss risks. The simulations can be performed with any number of peripherals connected to the DLU. They can start periodically or randomly, and are performed according to the following number of clock cycle probabilistic distributions:

- Probability mass function of the Poisson distribution, where x is a natural number, including 0, and $\lambda \in (0, +\infty)$ is the length. This type of distribution corresponds to a peripheral, which activates periodically around every λ clock cycle. The PMU’s periodic sampling of counters can be used as an example.

$$P(X = x) = \frac{\lambda^x}{x!} e^{-\lambda}. \quad (2)$$

- The probability mass function of the uniform discrete distribution, where x is a natural number in the interval $[a, b]$, a and b are natural numbers, and $n = b - a + 1$. This distribution corresponds to a peripheral that is expected to produce log data in an interval of clock cycles between a and b from its previous activation. The EMU and TMU (when triggered by an error) serve as examples of this kind of behavior.

$$P(X = x) = \frac{1}{n}. \quad (3)$$

- The probability mass function of Laplace distribution, where x is a real number, μ is the position of the distribution peak, and λ is the exponential decay (non-negative). This distribution corresponds to a peripheral that we expect to activate after a delay μ , and we can regulate the variability of this delay. For example, the ETU may be activated by software events that occur only after a fixed known delay.

$$P(X = x) = \frac{1}{2\lambda} e^{-\frac{|x-\mu|}{\lambda}}. \quad (4)$$

6.1. Real Use-Case for DLU

In this section, we show the analysis results that were obtained by using the set of peripherals described in Section 1. We considered a periodic activation for the PMU, as

well as a uniform distribution with a particularly large span (1 to 10 ms) for the EMU and TMU. The ETU was simulated with different distributions in both single- and multi-core solutions. We consider the DLU output capable of transferring data from the proxy's FIFO with a maximum rate of 640 Mbps and a mean value of nearly 40 Mbps (with a 20 MHz clock imposed by the targeted RISC-V architecture on Xilinx UltraScale+ FPGA).

The latency between the moment we start receiving data from the input peripheral and the end of transmission through the DLU's output mostly depends on the sizing of the proxy sections and the DLU buffer. Smaller sections and buffers reduce the latency but increase the risk of overflow.

The following list presents the main parameters to be tuned and the corresponding effects they have on the behavior of the log mechanism:

1. Number of cores: This corresponds to the number of ETUs connected to the DLU. It has an extreme influence on the quantity of data produced.
2. PMU sampling period: This regulates the periodicity of the data generated by the PMU. The higher the sampling frequency, the higher the buffers will be stressed.
3. PMU performance counters: These influence the quantity of data from the PMU.
4. EMU actions: These set the quantity of data from the EMU, but they do not have much influence on the log behavior. The EMU, like the TMU, is meant to rarely activate, and only if errors occur.
5. ETU annotation mode: Through this, one can choose a probabilistic distribution (like the ones listed above) and their respective parameters. This mode deeply influences the behavior of the system and has to be analyzed carefully.
6. DLU buffer sizes and sections: These are essential for the behavior of the DLU+Proxies. The current DLU's buffer only has two sections, but we are investigating the possibility of adding more sections. Every time a section of the DLU buffer is full, it is ready to be dumped and we can start filling the other one. This mechanism prevents data overwriting, and these parameters influence the average and max percentages of the buffers.
7. Proxy sections and subsections: For each proxy, we can set the number of sections and subsections. This is a way through which to regulate the mean and maximum values of the buffers with the aim of preventing data overflows.

Table 2 shows the values that are fixed for the simulations presented in this manuscript. While the clock frequency was imposed by RISC-V technology for UltraScale+ FPGA [21], other architectural choices were the results of a preliminary examination of the system.

Table 2. Constraints and sizings for the behavioral simulation.

Element	Sizing	Notes
Simulation Time	50 ms	Simulate 50 ms of log
Clock Frequency	20 MHz	PULPissimo Core Freq.
Performance Counter(s)	16	Maximum of counters
PMU Sampling Period	200 us	PMU counters sampling
Error Actions(s)	5	Reasonable number for recovery actions
TMU Buffer Size	$4 \times 5 \times 1 \times 3$ Byte	Equation (1) applied
EMU Buffer Size	$4 \times 5 \times 1 \times 16$ Byte	Equation (1) applied
ETU Buffer Size	$4 \times 5 \times 10 \times 3$ Byte	Equation (1) applied
PMU Buffer Size	$4 \times 5 \times 1 \times 35$ Byte	Equation (1) applied

6.2. Simulation Results

Figure 8 shows the maximum number of overflows completed by the ETU proxy, which was achieved by swapping from a single- to an octa-core implementation. In our

simulations, the DLU buffer size was equal to 2 KB, and we evaluated the behavior of the system by increasing the length λ of the Poisson distribution in one case by using Equation (4), and the width of the uniform distribution span (achieved by fixing the lower bound to 1) in the other by using Equation (3). The results highlighted that the performance highly depends on the structure of the software that is running. In particular, the uniform distribution requires a minimum of around 50 clock cycles in a worst-case scenario. Meanwhile, the Poisson distribution behaves correctly when there is a length of 30 clock cycles. Taking into account a typical real-time OS implementation, the average context switch and task annotation routines usually last more than 50 clock cycles (more than 2.5 μ s at 20 MHz). This means that our architecture can work in every condition and for each task's activation frequency.

Figure 9 shows the average utilization level of a 2 KB DLU buffer with a Poisson distribution, whereby there is an increasing number of cores. Compared with Poisson's data shown in Figure 8, the results highlight that a safe level for the DLU buffer is below 70% of the average utilization, and this is such that the protection from any boost in data production can be guaranteed. For the small λ values, the average of the DLU's buffer was stuck at nearly 73%, and the DLU was no longer capable of managing the data from the proxies, which were stuck at 100% and produced overflows. In these conditions, when a section is completely dumped, the DLU buffer's level drops immediately to 50%, but it rapidly reaches 100% again. This explains why the average closely approaches the midpoint value (75%) between these boundaries.

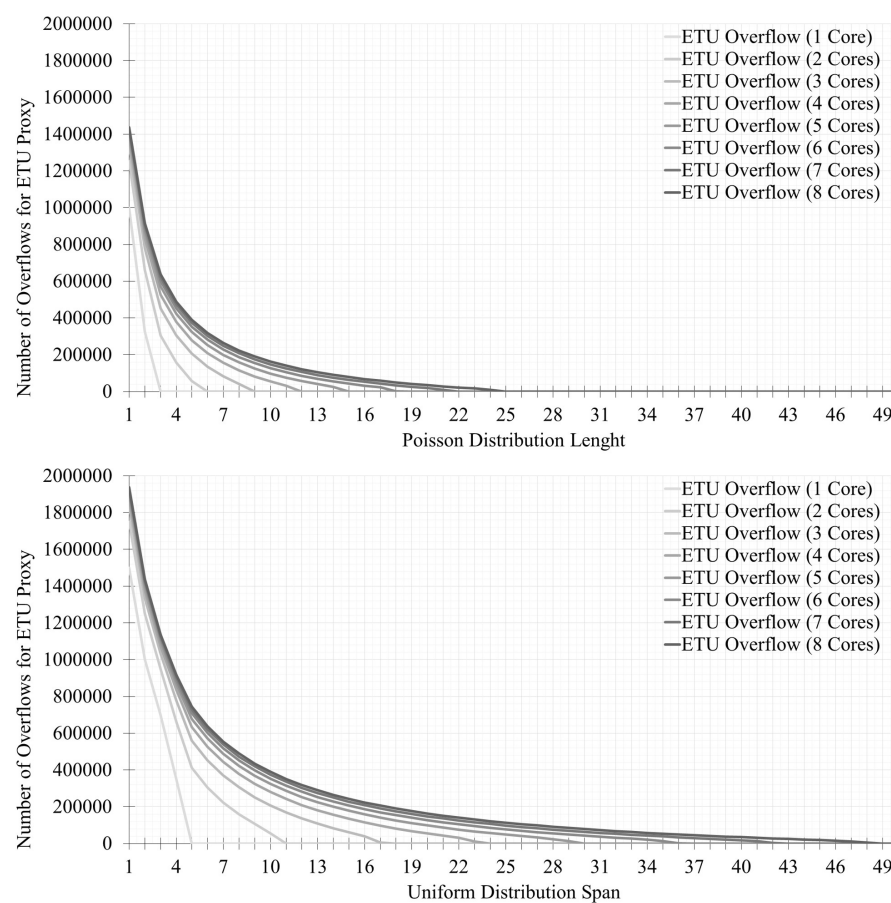


Figure 8. Comparison between the number of overflows in the ETU proxy for both Poisson and uniform distributions of the SW items' activation, ranging from a single-core to an octa-core implementation. Here, we noticed that the uniform distribution is more critical than the Poisson one. However, the minimum span required is in the order of a typical context switch for embedded applications.

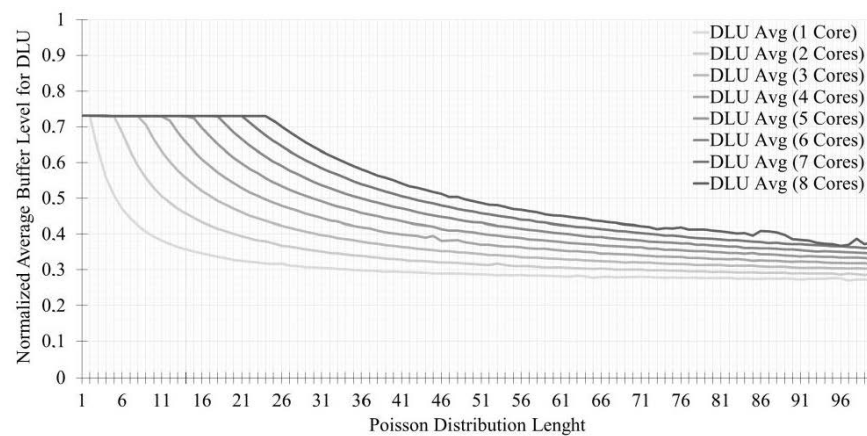


Figure 9. Comparison between the normalized average DLU's buffer levels for a Poisson distribution of the SW items' activation, ranging from a single-core to an octa-core implementation. The average was computed by considering the DLU's buffer filling percentage at every clock cycle of the simulation.

Once the safe conditions were considered, Figure 10 shows how to match the right size of the DLU buffer. In this graph, a single-core simulation was considered by increasing the size of the log buffer. The graphs of the DLU max slowly decreased while increasing the Poisson's Length. The overshoot above 50% (which was a section that was completely filled) was constantly reduced. The presented data for the different buffer sizes tended to show the same behavior.

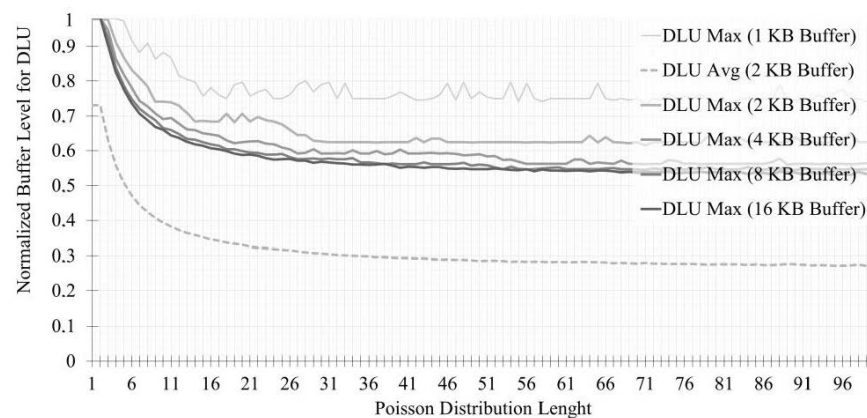


Figure 10. Comparison between the normalized DLU's buffer levels (for a 2 KB size and the average value) for a Poisson distribution of the SW items' activation from a 1 KB to a 16 KB DLU buffer size.

The only trade-off we have is to guarantee a safe margin and a tolerable area/power overhead. For example, in a single-core design, a 2 KB buffer is enough to manage even the fastest software items' activation (three instructions, corresponding to three clock cycles).

From these results, we obtained the implementation described in Section 7.

7. DLU Hardware Synthesis and Implementation

In this section, we discuss the results of a real implementation of PULPissimo and safety peripherals plus DLU hardware for a Xilinx Zynq ZCU102 UltraScale+ (xczu9eg-ffvb1156-2-e) when using Vivado 2018. Table 3 presents all of the constraints imposed by the system, the safety peripheral sizes, and the proxy and DLU buffer sizes, which were obtained by means of the simulations described in Section 6.

We sized the peripherals with the maximum number of performance counters (16) and a reasonable number of error actions (5). Then, the number of cores (single-core) and the number of safety peripherals determined the number of DLU ports. The buffers were sized according to the analysis performed in Section 6. We implemented a 2 KB buffer for the

DLU, and more than 1.5 KB for the peripheral proxy buffers. If we consider the implemented memory to be exclusively dedicated to the log, it is comparable to a memory space of 3.64 KB.

Table 3. Sizes of the peripherals, internal buffers, and the processing unit.

Element	Sizing	Notes
Core(s)	1	PULPissimo single core unit
Core Frequency	20 MHz	Core and Safety Peripheral clock
SoC Frequency	10 MHz	SoC clock
Performance Counter(s)	16	PMU performance counters
Error Actions(s)	5	EMU possible actions
DLU Port(s)	4	Peripherals connected to DLU
DLU Buffer Size	$4 \times 2 \times 256$ Byte	2 KB of memory
TMU Buffer Size	$4 \times 5 \times 1 \times 3$ Byte	60 Byte of memory
EMU Buffer Size	$4 \times 5 \times 1 \times 16$ Byte	320 Byte of memory
ETU Buffer Size	$4 \times 5 \times 10 \times 3$ Byte	600 Byte of memory
PMU Buffer Size	$4 \times 5 \times 1 \times 35$ Byte	700 Byte of memory

The top level of the safety peripherals plus DLU was the advanced peripheral bus (APB), which makes the registers accessible by PULPissimo. The lowest level is composed of the DLU, the proxies, and the peripherals (with all their sub-elements). Table 4 shows the device costs in terms of logic elements. Only 9.70% of the employed CLB LUTs were for execution tracing and logging, and the CLB registers for the DLU and buffers were 7.18% of those implemented. Meanwhile, CARRY8 was used for tracing, and it occupied more than 20% of the employed logic (23.65%); likewise, a similar value was reached by the CLBs for the combinatory logic (16.64%).

Table 4. The Vivado post-implementation analysis of the logic utilization. These are the top three levels of the hierarchy of the safety peripherals.

HW Level	CLB LUTs	CLB Reg	CARRY8	F7 and F8 Muxes	CLB Logic	BUFG
APB Periph Interface	6752	12,111	812	403	2025	4
Safety Periph plus DLU	5037	9423	812	16	1853	4
EMU	350	377	24	0	124	0
Proxy EMU	149	185	46	0	69	0
ETU	16	193	2	0	41	0
Proxy ETU	154	192	46	0	73	0
PMU	3975	7838	582	16	1383	4
Proxy PMU	158	192	46	0	77	0
TMU	41	145	12	0	43	0
Proxy TMU	153	192	46	0	74	0
DLU	41	109	8	0	44	0

CLB LUT: Configurable logic block used as a look-up table. CLB Reg: The CLB used as a register. CARRY8: Fast carry logic with a look ahead. F7 Muxes: The CLB multiplexer to connect the look-up table together. F8 Muxes: The CLB multiplexer to connect the F7 muxes together. CLB Logic: The CLB used as logical circuits. BUFG: global clock buffers.

From the obtained results, it is clear that only a portion of the logic introduced by the safety peripherals is dedicated to the log functionalities. The combined size of the safety peripherals plus DLU is less than 30% of a single-core PULPissimo implementation. Our devices increased the look-up tables by +14.5%, the number of flip-flops by +30.0%, and the clock's buffers by +33.4%. Moreover, if we apply the same logic to a multi-core environment, the area overhead will increase in terms of absolute values, but the ratio will reduce. Regarding the octa-core device, the percentages of the new logic introduced dropped to nearly +3% for the look-up tables and +5% for the flip-flops and BUFGs. We can consider increasing the log buffer size in order to ensure a greater covering margin for data overproduction.

Regarding the timing performances, the safety peripherals plus DLU architecture can work up to a clock frequency that is equal to 125 MHz. However, since the DLU does not have strict real-time requirements, a 20 MHz clock frequency (the same domain of the RI5CY core) was enough for our applications.

Moreover, the total on-chip power consumed by the RISC-V core and safety peripherals plus DLU was nearly 1 Watt. However, 70% of the consumption was from device static power consumption, which is commonly known to be high in FPGA devices. If we take into account the logic utilizations detailed above, we can assume that only a small portion of such consumption is, in reality, affected by the DLU and the other custom peripherals. The remaining 30% (0.29 Watt) of the on-chip power is due to dynamic consumption (clocks, signals, logic, and I/O), which can be entirely attributed to the implemented devices and RISC-V core. The implemented peripherals have a dynamic power consumption of 0.029 Watt (10% of the total), and, of these, 0.017 Watts are exclusively for the DLU and the proxies.

In conclusion, for an integrated implementation comprising both the DLU and proxies, we anticipate a power requirement of less than 0.05 Watts if we consider both the dynamic consumption and static power consumption.

8. Conclusions

In this manuscript, we present a complete hardware log system based on a central device (DLU) and some slave appendices (proxies). In Section 2, we show that the current state-of-the-art logging mechanisms address different problems in multiple different ways. In particular, they mainly focus on the collection of sensor outputs and environmental data, and they are used many times through the utilization of computational and other shared resources. If we compare the values obtained—in terms of power, area, and timing performance—with the ones posed in Section 2, our device results are in line with the discussed state-of-the-art technologies. Moreover, the DLU adds functionalities, which are not fully considered in the related works. The implemented device receives and efficiently manages the data from heterogeneous sources and generates a unique log, which is available for further timing analysis and fault inspection. Similarly, the same can be declared for software solutions that need specific tasks that run on CPUs to collect and transmit execution data and performance statistics. This, of course, produces interference and alters the results obtained for the system under inspection.

The main purpose of this manuscript was to study and implement an alternative method for system execution data acquisition, specifically one that overcomes SW solutions. Despite the disadvantages and constraints that are derived from having an HW-accelerated log feature, our approach allows one to free the MCU from absolving log and communication functionalities (i.e., Ethernet), thereby reducing the interference between traced items and the tracer itself. This allows one to have an accurate set of data for the execution, for the cutting-off of unpredictability, and for permitting precise estimations of WCET and FFI safety analyses.

Moreover, the hardware design and behavioral simulations allow us to determine the optimal sizes of devices involved in logging, specifically tailored to the desired specifications and performance. The implemented design for a single-core RISC-V SoC resulted in

reduced area and power overhead. Furthermore, regarding timing performance, the DLU can simultaneously log data from an octa-core microcontroller device. This confirms the fact that we can still allocate more resources to any other improvement that we would like to add to the current solution.

Author Contributions: Conceptualization, F.C.; methodology, F.C., P.G. and S.S.; software, F.C.; validation, A.A., P.G. and S.S.; formal analysis, F.C.; investigation, F.C.; resources, F.C.; data curation, F.C.; writing—original draft preparation, F.C.; writing—review and editing, A.A., P.G. and S.S.; visualization, F.C.; supervision, A.A., P.G. and S.S.; project administration, P.G. and S.S.; funding acquisition, S.S. and P.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data are available upon request due to restrictions. The data presented in this study are available upon request from the corresponding author. The data are not publicly available due to internal company regulations.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ismail, A.; Jung, W. Research Trends in Automotive Functional Safety. In Proceedings of the 2013 International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering (QR2MSE), Chengdu, China, 15–18 July 2013.
2. Synopsis ASIL. Available online: <http://www.synopsys.com/automotive/what-is-asil.html> (accessed on 7 May 2023).
3. Crespo, A.; Balbastre, P.; Simó, J.; Coronel, J.; Pérez, D.G.; Bonnot, P. Hypervisor-Based Multicore Feedback Control of Mixed-Criticality Systems. *IEEE Access* **2018**, *6*, 50627–50640. [CrossRef]
4. Christie, D.S. Microprocessor with Built-In Instruction Tracing Capability, US5944841A. 1997. Available online: <https://patents.google.com/patent/US5944841> (accessed on 10 June 2023).
5. Vaj-Troniks, E.; Dzhordan Lourens, M.L. Device for Recording and Transmitting Data of Movable Property Object, RU2639013C1. 2017. Available online: <https://patents.google.com/patent/RU2639013C1/en> (accessed on 10 June 2023).
6. Neff, A.; Thomas Koenigseder, T.W. Data Logging or Stimulation in Automotive Ethernet Networks Using the Vehicle Infrastructure, US9912531B2. 2018. Available online: <https://patents.google.com/patent/US9912531B2> (accessed on 10 June 2023).
7. Movable Property Data Logger and Transmitter, CN104520674B. 2019. Available online: <https://patents.google.com/patent/CN104520674B/en> (accessed on 10 June 2023).
8. GitHub, fpga-log. Available online: <https://github.com/carstenbru/fpga-log> (accessed on 10 June 2023).
9. Krebs, M.C.; Stamm, C. Active Data Logger. *MVS Fokus Rep.* **2011**, *5*, 15–21. Available online: <http://hdl.handle.net/11654/23489> (accessed on 13 June 2023).
10. Purwadi, A.; Haroen, Y.; Ali, F.Y.; Heryana, N.; Nurafiat, D.; Assegaf, A. Prototype development of a Low Cost data logger for PV based LED Street Lighting System. In Proceedings of the 2011 International Conference on Electrical Engineering and Informatics, Bandung, Indonesia, 17–19 July 2011; pp. 1–5. [CrossRef]
11. Kandadai, V.; Sridharan, M.; Manickavasagam Parvathy, S.; Pitchaimuthu, R. A comprehensive embedded solution for data acquisition and communication using FPGA. *J. Appl. Res. Technol.* **2017**, *15*, 45–53. [CrossRef]
12. Basic, F.; Steger, C.; Kofler, R. Embedded Platform Patterns for Distributed and Secure Logging. In Proceedings of the 26th European Conference on Pattern Languages of Programs, New York, NY, USA, 7–11 July 2021; EuroPLoP’21. [CrossRef]
13. Lousa Marques, A. Monitoring Architecture for Real Time Systems. Master’s Thesis, Universidade da Beira Interior, Covilhã, Portugal, 2016.
14. TraceCompass. Available online: www.eclipse.org/tracecompass/ (accessed on 7 May 2023).
15. Segger SystemView. Available online: www.segger.com/products/development-tools/systemview (accessed on 7 May 2023).
16. Percepio Tracealyzer. Available online: www.percepio.com/tracealyzer/ (accessed on 7 May 2023).
17. Rapitask. Available online: www.rapitasystems.com/products/rapitask (accessed on 7 May 2023).
18. Gliwa T1. Available online: www.gliwa.com (accessed on 7 May 2023).
19. Lauterbach. Available online: www.lauterbach.com (accessed on 7 May 2023).
20. RISC-V. Available online: <https://riscv.org/about/> (accessed on 7 May 2023).
21. Pullini, A.; Rossi, D.; Loi, I.; Tagliavini, G.; Benini, L. Mr.Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for IoT Edge Processing. *IEEE J. Solid-State Circ.* **2019**, *54*, 1970–1981. [CrossRef]
22. Kurth, A.; Vogel, P.; Capotondi, A.; Marongiu, A.; Benini, L. HERO: Heterogeneous Embedded Research Platform for Exploring RISC-V Manycore Accelerators on FPGA. In Proceedings of the CARRV’17, Boston, MA, USA, 14 October 2017.
23. Balkind, J.; McKeown, M.; Fu, Y.; Nguyen, T.; Zhou, Y.; Lavrov, A.; Shahrad, M.; Fuchs, A.; Payne, S.; Liang, X.; et al. OpenPiton: An Open Source Manycore Research Framework. In Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, Atlanta, GA, USA, 2–6 April 2016.

24. Special Interest Group—Safety RISC-V. 2021. Available online: https://www.hpcuserforum.com/wp-content/uploads/2021/09/RISC-V_J.Davis-J.Leidel_Sept-2021-HPC-UF.pdf (accessed on 8 November 2021).
25. Cosimi, F.; Tronci, F.; Saponara, S.; Gai, P. Analysis, Hardware Specification and Design of a Programmable Performance Monitoring Unit (PPMU) for RISC-V ECUs. In Proceedings of the 2022 IEEE International Conference on Smart Computing (SMARTCOMP), Helsinki, Finland, 20–24 June 2022; pp. 213–218.
26. Vilardell, S.; Serra, I.; Santalla, R.; Mezzetti, E.; Abella, J.; Cazorla, F.J. HRM: Merging Hardware Event Monitors for Improved Timing Analysis of Complex MPSoCs. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 3662–3673. [[CrossRef](#)]
27. Cosimi, F.; Tronci, F.; Saponara, S.; Gai, P. Analysis, Design and Synthesis of an Execution Tracing Unit (ETU) Based on AUTOSAR Run-Time Interface (ARTI). In Proceedings of the International Conference on Applications in Electronics Pervading Industry, Environment and Society, Genoa, Italy, 26–27 September 2022; pp. 82–89.
28. Bo, H.; Hui, D.; Dafang, W.; Guifan, Z. Basic Concepts on AUTOSAR Development. In Proceedings of the 2010 International Conference on Intelligent Computation Technology and Automation, Changsha, China, 11–12 May 2010; pp. 871–873.
29. ARTI. Available online: [AUTOSAR_SWS_ClassicPlatformARTI.pdf](#) (accessed on 7 June 2022).
30. Forencich, A.; Snoeren, A.C.; Porter, G.; Papen, G. Corundum: An Open-Source 100-Gbps Nic. In Proceedings of the 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Fayetteville, AR, USA, 3–6 May 2020; pp. 38–46. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.