WP540 (v1.0)

Kria Robotics Stack

A ROS 2-centric Approach for Hardware Acceleration in Robotics

By: Víctor Mayoral-Vilches

Hardware acceleration is revolutionizing robotics—empowering robots with the ability to react faster, consume less power, and present more secure architectures. Enter the future of robot chips built with ROS 2 as its SDK and specialized robotic circuitry built with SoCs.

ABSTRACT

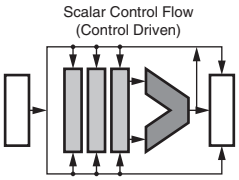
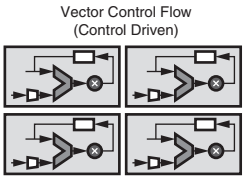
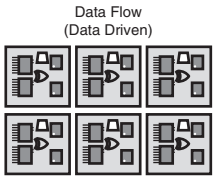
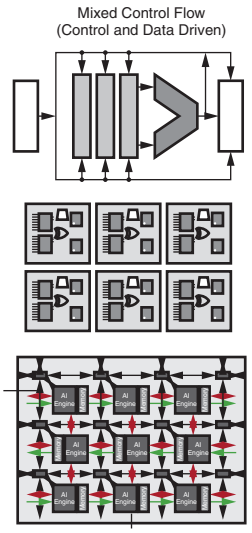
The Kria™ robotics stack (KRS) is an integrated set of robot libraries and utilities that use hardware to accelerate the development, maintenance, and commercialization of industrial-grade robotic solutions. It adopts ROS 2 as the Software Development Kit (SDK) and proposes a ROS 2-centric development approach that spans from the creation of computational graphs to the commercialization of ROS 2 overlay workspaces found in the Xilinx® App Store. Specifically targeting Kria SOMs, KRS delivers a production-ready, high throughput, low latency, and real-time value to robotics. It gives roboticists the capability to create custom, secure compute architectures, and the option to commercialize them through the Xilinx App Store. KRS is the entry point for ROS 2 roboticists into the world of hardware acceleration.

Introduction

With the decline of Moore's Law, specialized computing units capable of hardware acceleration have proven to be the answer for achieving higher performance in robotics [Ref 1]. Such specialized computing architectures rely on specific hardware (i.e., through adaptive System-on-Chips), tailored to the robotic algorithm for obtaining “fast computation" (getting things done quickly once started) and "real time" (meeting the time deadlines set for each task) behaviors. This is the result of *hardware acceleration*. The core idea is to combine traditional control-driven approach used in robotics, with the data-driven approach optimizing the amount of hardware resources and, as a consequence, the performance [Ref 2]. In addition, it allows more power-efficient, deterministic and secure hardware structures.

Table 1 compares different computational models in robotics and shows how the combined control-and-data driven model provides the best trade-off. However, the process of creating specialized hardware comes at the cost of complexity.

Table 1: **Comparison of Hardware Computational Models in Robotics**

Computational Model	Scalar Control Flow	Vector Control Flow	Data Flow	Mixed (Control and Date Driven)
				
Technology	**CPUs**	**DSPs, GPUs, and AI Engines**	**FPGA**	**Adaptive SoCs (CPUs + FPGAs), ACAPs (CPUs + FPGAs + AI Engines)**
Definition	Scalar Von-Neumann processor. Conventional computing. A token of control indicates when a statement should be executed.	Vector Von-Neumann processor. Token of control for executing a vectorized single instruction. Fixed processing and memory architectures.	Flexible processing structures. Eager evaluation; statements are executed as soon as data is available.	Adaptive processing structures. Capability to build mixed control and data-driven compute models mixing different processors.
Programming Capabilities	SW-programmable	SW-programmable	HW-level programmable	SW-programmable and HW-level programmable
Advantages	1. Full control 2. Complex data and control structures are easily implemented.	1. Full control, 2. Domain-specific parallelism (e.g., math, video, and image processing)	1. Very high potential for parallelism 2. High throughput 3. Deterministic: Free from side effects	1. Full control capabilities 2. Complex data and control structures easily implemented 3. Very high potential for parallelism 4. High throughput 5. Deterministic 6. Domain-specific parallelism
Disadvantages	1. Less efficient in managing asynchronous events (interrupts alter computational cycles) 2. Limited parallelism scalability (# of CPU cores) and execution capability due to fixed control path	1. Fixed memory and compute architectures (efficiency bottlenecks) 2. GPUs cannot operate directly on real-time data streams typical of physical systems like robots 3. DSP implement specialized instructions, but suffer of the same trade-off frequency/performance	1. Long development cycle 2. Difficult to program and architect 3. Data flow is privileged versus control flow	1. Need allocation (often manual) of different tasks into different computing units to exploit performance 2. Require coordination among the different computing units (when, how, and where get the data)

Complexity is not novel in robotics. On average, it takes one to two months for a PhD student to become familiar with the basic concepts, and years to build complex computational graphs with the robot operating system (ROS) [Ref 3], the de facto standard for robot application development. The importance of ROS, as well as past attempts of using hardware acceleration with it, are covered in *Adaptive Computing in Robotics, Leveraging ROS 2 Enabled Software-defined Hardware for FPGAs* (WP537) [Ref 2].

Beyond its complexity, another aspect to account for when building robot architectures is real-time. Robots are inherently deterministic machines. They are networks of networks. Altogether, these networks are understood as the nervous system of the robot, which facilitates exchanging information in a timely deterministic manner. Like in the human nervous system, real-time across all these networks is fundamental for the robot to behave coherently. Real-time should, therefore, be guaranteed at the intra-process (robot computation nodes within the same process), inter-process (computation nodes between processes in the same computer), and intra-network levels. CPU and GPU fixed Von-Neumann based architectures excel in control flow but struggle to guarantee determinism. Moreover, architects using these fixed compute substrates generally have to choose between high throughput or low latency.

This white paper introduces the Kria™ robotics stack (KRS), a set of libraries and utilities for robotics tightly integrated with ROS 2 to accelerate the development, maintenance, and commercialization of industrial-grade robotic solutions. KRS is designed for delivering low latency (fast computation), determinism (predictable), real-time (on time), security, and high throughput thorough hardware acceleration. It provides binding structures into ROS 2, enabling programs that execute across heterogeneous platforms for CPUs and GPUs, including FPGA high-level synthesis (HLS) for compute acceleration. It also leverages the Kria SOM portfolio and the reference development boards as platforms to create custom and secure computing architectures.

KRS is built to facilitate hardware acceleration in ROS 2 while remaining conscious of the real-time needs that robots have across their networks. Prior work investigating ROS 2 identified time bottlenecks and limitations across the OSI stack [Ref 4] [Ref 5] [Ref 6] [Ref 7] [Ref 8]. To deliver all of this together and ensure determinism in distributed ROS 2 computational graphs, KRS components come in three categories, allowing architects to select the components needed to meet the requirements of their application:

- **ROS 2 applications and libraries**: This group corresponds with acceleration kernels that speed-up OSI L7 applications or libraries on top of ROS 2. Any computation on top of ROS 2 is a good candidate for this category. Examples include selected components in the navigation, manipulation, perception or control stacks, as depicted in Figure 1.
- **ROS 2 core**: This includes kernels that accelerate or offload OSI L7 ROS 2 components and tools to dedicated robot circuitry (e.g., in an FPGA or an adaptive SoC), with focus on `rclcpp`, `rcl`, `rmw`, and the corresponding `rmw_adapters`, for each supported communication middleware.(1) Examples include FPGA-offloaded ROS 2 executors for more deterministic behaviors [Ref 9][Ref 10], or complete hardware offloaded ROS 2 nodes [Ref 11].
- **ROS 2 underlayers**: This category groups together all accelerators below the ROS 2 core layers belonging to OSI L2-L7, including communication middleware. Examples of these types of accelerators include a complete or partial FPGA-accelerated DDS implementation, an offloaded networking stack, or a data link layer for real-time deterministic, low latency, and high throughput interactions.

1. The communication middleware itself is not considered part of the ROS 2 core accelerators category. The rationale behind this is to remain coherent with the ROS 2 design principles of remaining agnostic to the underlying communication middleware. This way, any arbitrary communication middleware (e.g., DDS) will belong to the next category, the ROS 2 underlayers.

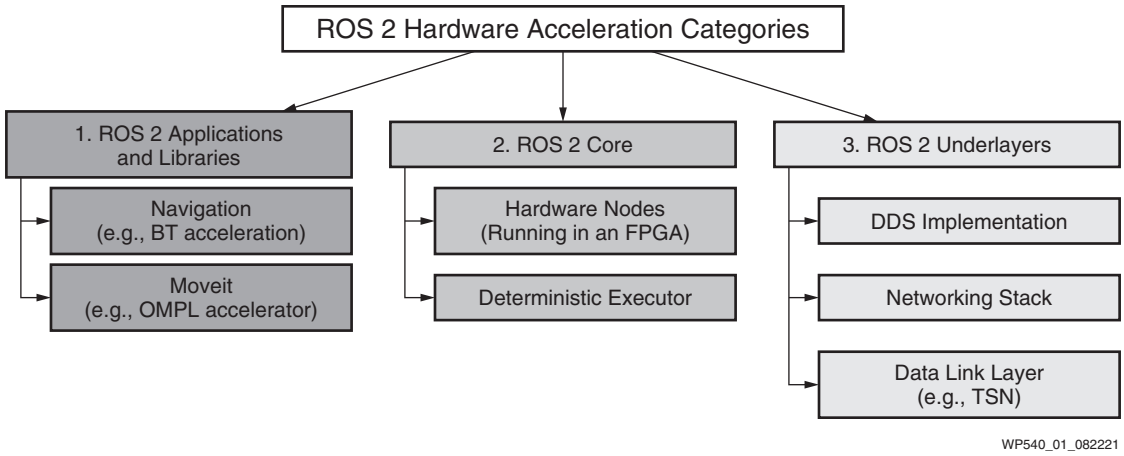


Figure 1: **ROS 2 Hardware Acceleration Paths and Examples**

The remaining content of this white paper is organized as follows. The Kria Robotics Stack section describes KRS in more detail, including design choices and capabilities. It includes: the ROS 2-centric nature, the KRS mixed source ecosystem to facilitate the commercialization of ROS 2 accelerators, and real-time considerations. Use Cases presents two KRS use cases: Accelerating ROS 2 applications and libraries, and optimizing ROS 2 underlayers for determinism. Xilinx App Store describes in more detail how KRS empowers roboticists to protect and monetize their accelerated ROS 2 packages.

Kria Robotics Stack

KRS is a ROS 2 superset for industry. It delivers tools and robot libraries that help create software-defined robotics solutions. It bridges the world between roboticists and Xilinx, simplifying the use of hardware acceleration. KRS divides its components into five groups: application and artificial intelligence, middleware, OS and hypervisor, silicon architecture, and safety[1] and security.[2] Figure 2 presents the stack. The following subsections cover the most relevant design choices around KRS.

1. https://www.xilinx.com/publications/solution-briefs/xilinx-functional-safety-solution-brief.pdf
2. For more on safety and security, refer to [Ref 12][Ref 13].

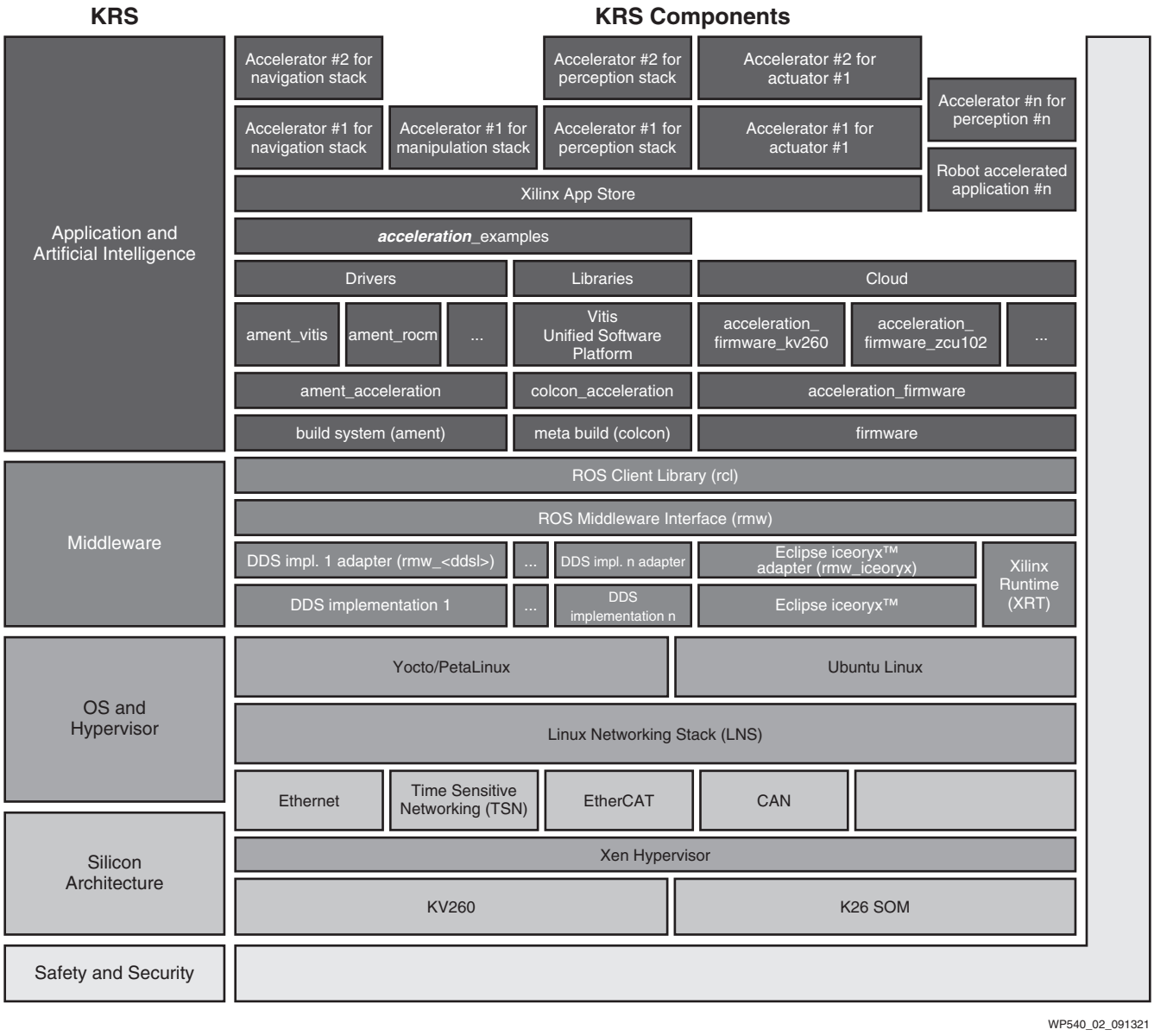


Figure 2: **Kria Robotics Stack Tools and Components**

A ROS 2-centric Approach, for All Roboticists

ROS is to roboticists what Linux is to most computer scientists and software developers. ROS 2 increases ROS capabilities, improving the robot's behavior for production-ready systems. Without reinventing the wheel with approaches that overload the space with replicas of libraries and similar simulators, KRS meets the ROS robotics community interests and builds on top of ROS 2, together with its tightly connected robotics simulator, Gazebo [Ref 14].

Figure 2 depicts how the ROS 2 layers (`rcl`, `rmw`, etc.) sit at the core of KRS architecture, within the middleware group, surrounded by layers that enhance ROS 2 capabilities to be more deterministic and deliver higher performance across robotic interactions.

Part of the Xilinx commitment to the ROS 2 ecosystem includes the creation of the ROS 2 Hardware Acceleration Working Group (HAWG), spearheaded by Xilinx. This working group aims to drive the

creation, maintenance, and testing of acceleration kernels on top of open standards (C++ and OpenCL) for optimized ROS 2 and Gazebo interactions over different compute substrates (including FPGAs, adaptive SoCs, and GPUs). As specified on the community announcement, the group targets hardware acceleration in:

- Embedded (edge) devices
- Workstations
- Data centers
- Cloud

Initially, the plan is to demonstrate acceleration on edge devices using simple examples at the application layer that roboticists can use as a blueprint for their designs. HAWG then will target underlayers to optimize interactions between nodes within the ROS 2 computational graph. After this, the group will pivot into kernels that accelerate the application level, with initial consideration for perception, actuation, navigation, and manipulation.

Figure 3 illustrates the ROS 2 enhancements contributed by Xilinx to the HAWG. This work integrates hardware acceleration at the core of the ROS 2 build system (`ament`) and meta build tools (`colcon`) in a technology agnostic manner, supporting future extensions with other technologies including AI Engines and GPUs. The process of generating specialized robotic circuitry can be controlled and triggered directly from ROS 2 workspaces using `colcon` with specific flags that automate cross-compilation of host code, synthesis, and implementation of accelerators. ROS 2 robotic custom hardware accelerators are, therefore, generated without the need to interact with vendor specific tools (e.g., Vitis® or Vivado® tools). The resulting kernels are contained within a ROS 2 overlay workspace, which can be packaged and shipped directly into the robot.

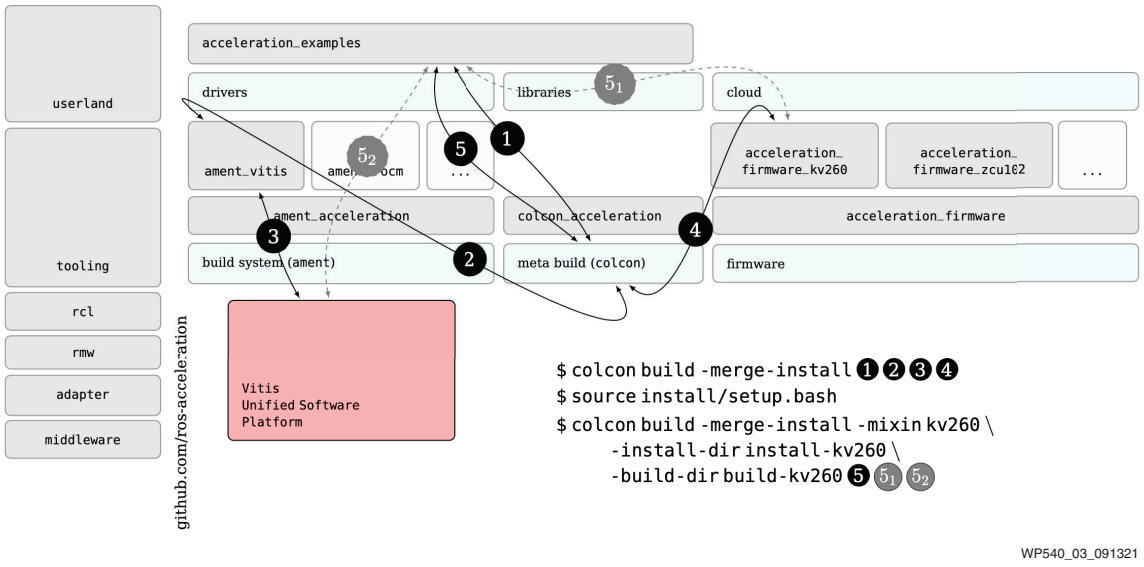


Figure 3: **Interaction with ROS 2 Packages in the HAWG's Initial Architecture**

Through the `ament_vitis` ROS 2 package, roboticists have a familiar software driven flow to implement accelerators. Xilinx's technology improves productivity using three build targets, two of which correspond with QEMU emulations for accelerated development:

- **Software Emulation (`sw_emu`):** The kernel code is compiled to run on the host processor. This allows iterative algorithm refinement through fast build-and-run loops. This target is useful for identifying syntax errors, performing source-level debugging of the kernel source code while running with the application, and verifying the functional behavior of the system. This build target provides a transformation that runs all the code in an emulated processor matching the target hardware (e.g., KV260), as if there were no accelerator.
- **Hardware Emulation (`hw_emu`):** The kernel code is compiled into a hardware model, which is then run in a dedicated simulator. This build-and-run loop takes longer, but it provides a detailed, cycle-accurate view of the kernel activity. This is useful for testing the functionality of the programmable logic that goes into the adaptive SoC and supplying initial performance estimates. It provides a simulation of the adaptive SoC in an emulation of the target hardware (e.g., KV260).
- **Hardware (`hw`):** The kernel code is compiled into a hardware model and then implemented for the targeted adaptive SoC, resulting in a binary that runs on the actual adaptive SoC.

Some of the ROS 2 package examples provided in the `acceleration_examples` meta-package, e.g., `sw_emu`, take approximately one minute to build. The `hw_emu` takes four minutes, and the `hw` takes up to ~23 minutes.(1)

Although KRS is built using ROS 2 as its SDK, to serve all roboticists, KRS also empowers alternative flows using other robotic frameworks, or no robotic framework at all (e.g., by directly using the Xilinx Vitis Integrated Software Platform).

Mixed Source Ecosystem to Boost Professional Use

The ROS ecosystem brings together a worldwide community of thousands of roboticists developing robot applications while using ROS abstractions [Ref 2]. In a way, ROS is the common API roboticists use when building robot behaviors, the reference SDK in robotics. With the advent of mixed source technology ecosystems in robotics [Ref 15], there are already various examples of companies providing value around open-source packages, while contributing to the community. Figure 4 presents the mixed-source ecosystem proposed by KRS and depicts various contributions already made by Xilinx.

1. Builds performed on a workstation using an AMD Ryzen 5 PRO 4650G.

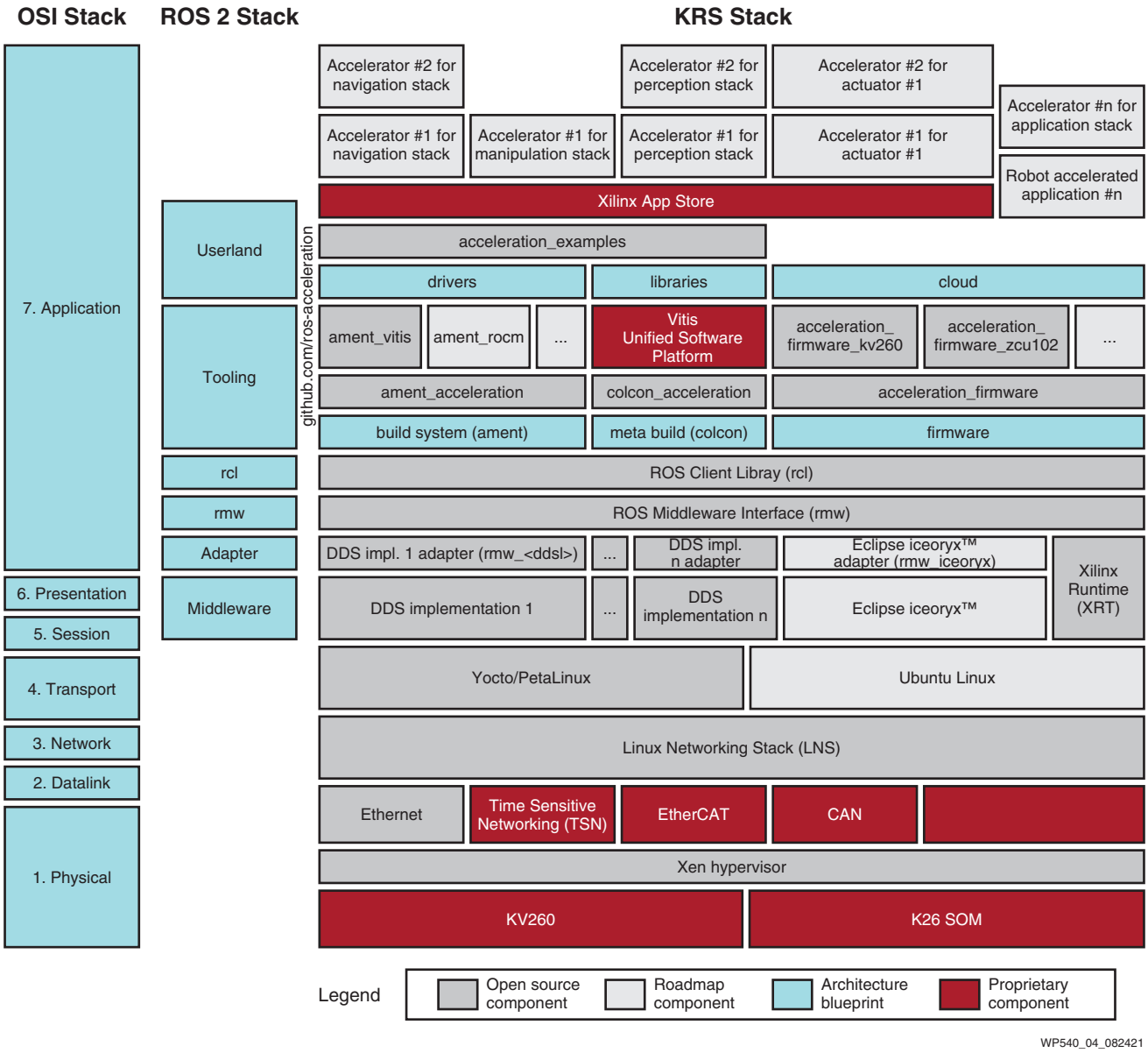


Figure 4: **KRS Mixed Source Ecosystem in Relation to the OSI and ROS 2 Stacks**

KRS proposes a layer for commercialization of ROS robot accelerated applications. By leveraging Xilinx's experience using encryption and authentication to secure FPGA bitstreams, the Xilinx App Store proposes a managed, easy to use, and secure infrastructure for digital rights management (DRM) where the applications can be commercialize and discovered by global customers. Through the Xilinx App Store connection, KRS containerizes ROS 2 overlay workspaces into robot accelerated applications, allowing selected partners to protect and monetize their accelerated ROS 2 packages.

Real-time at the Core of KRS

Real-time is an end-to-end characteristic of a robotic system. A ROS 2 application running in a scalar processor (i.e., a CPU) suffers from different sources of indeterminism. Figure 5 depicts these across the OSI stack. For a robot to respond in a deterministic manner(1), all layers across the stack (L1-L7) should respond coherently. For each one of these levels, the sources of indeterminism need to be addressed.

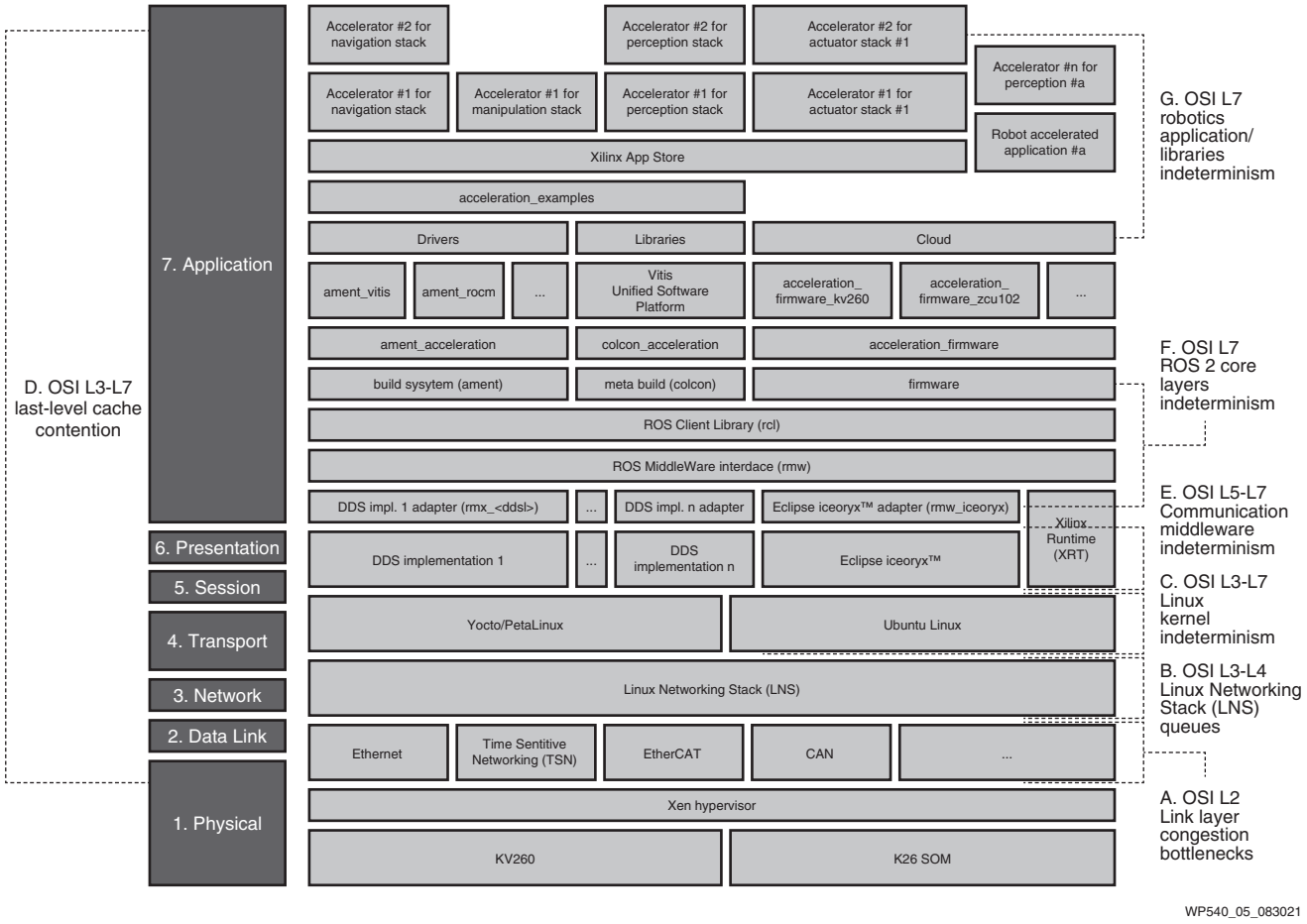


Figure 5: **Sources Of Indeterminism in the ROS 2 Stack**

In general, solutions to real-time problems in CPUs fall into two big categories: 1) Setting the correct priority in the corresponding abstraction, and 2) Applying quality of service (QoS) techniques. Each layer has its own QoS methods. Layer 2 uses the QoS techniques specified in the existing IEEE Std 802.1Q standards, and new techniques such as the time sensitive network (TSN) standards. For the Linux network stack, traffic control is used to configure QoS methods. Similarly, from the Linux kernel all the way up to the application libraries, each layer needs to be configured for bounded maximum latencies for the robotic system to deliver real-time capabilities.

The goal of KRS is to provide mechanisms to mitigate all these indeterminism issues through a modular approach. Selected modules can be prioritized and used to remove the desired sources of indeterminism, adapting time mitigations to each use case. The following sections analyze the

1. Note the difference between real-time (deterministic, on time) and fast computation (low latency).

indeterminism sources and provide a short description around the solutions integrated within KRS. They are labeled (e.g., A, B, C) to correspond with the callouts in Figure 5):

A. Link Layer Congestion Bottlenecks: Real-time problems related to data link-layer bottlenecks (OSI L2) are a relevant source of indeterminism in robotic systems. The most common scenario is when different traffic sources are present in the same network segment with different orders of priority. For example, congestion problems can arise when various depth sensors generating point clouds of the environment are transmitted over switched Ethernet networks, and mixed together with higher priority traffic (e.g., that commands actuators). This can lead to undesired indeterminism or even to packet drops for time critical packages. KRS addresses these problems by providing time sensitive link-layer components (such as TSN), which enable deterministic real-time communications over Ethernet.

Another source of indeterminism in the data link layer comes while interoperating between different communication buses, e.g., relaying the sensing data perceived over Ethernet to an EtherCAT field bus, or passing annotated data from Ethernet to CAN. The interfacing between these networks in scalar processors typically happens at higher layers (e.g., OSI L7), which besides consuming many additional processing cycles, exposes data to the additional indeterminism of upper abstraction layers. By leveraging the adaptive SoC, KRS provides capabilities to create specialized circuitry to bridge communications between these networks while operating on the data, everything while at the data link layer.

Figure 5 shows the sources of indeterminism in the ROS 2 stack. KRS provides mechanisms to mitigate these sources of unbounded latency for robotic applications.

B. Linux Networking Stack (LNS) Queues Congestion: The real-time problems in the LNS arise when networking resources (e.g., queues) that are shared between ROS 2 publishers and subscribers in the same device cause congestion issues, leading to undesired latencies. These delays become especially apparent when considering the threads processing the networking messages, which might be competing for CPU time (further enhancing the real-time issues in the Linux kernel indeterminism). KRS leverages the Linux kernel traffic control (TC) capabilities to mitigate the indeterminism of the LNS.

C. Linux Kernel Indeterminism: This source accounts for the real-time problems related with the indeterminism inside the operating system (OS) or more specifically, the lack of timely preemptions at the kernel level, including its drivers. For example, the lack of preemption capabilities in a kernel driver can lead to communication interactions that suffer delays because CPU cycles arrive late when processing, delivering, or retrieving messages. KRS addresses these real-time issues by offering a fully preemptible Linux kernel (`PREEMPT_RT`) and by maintaining a Xilinx-specific Linux kernel fork with drivers optimized for determinism and preemption.(1)

D. Last Level Cache (LLC) Contention: Another large source of real-time interference in multi-core scalar processors is the last level cache. In a multicore CPUs, the LLC is shared by all cores. This means that a noisy neighbor can have a drastic impact on the worst-case execution time (WCET) of software on other cores. Past research [Ref 16] showed a slowdown factor of 340 times, swamping the theoretical four times speedup expected from a quad-core multicore processor. KRS addresses this problem by cache coloring at the Xen hypervisor level. Cache coloring consists of partitioning the L2 cache and allocating a cache entry for each virtual machine (VM) in the hypervisor [Ref 17].

1. See https://github.com/Xilinx/linux-xlnx/

This approach allows real-time applications to run deterministic IRQs, lower the time variance, reduce the effects of cache interference, and allow mixed critical workloads on a single adaptive SoC.

E. Communication Middleware Indeterminism: Often architected and programmed with a data connectivity and throughput mindset (as opposed to determinism), the communication middleware also represents a source of indeterminism. Real-time safe coding practices should be applied when implementing the communication middleware, and also when using in higher abstraction layers. In addition, consider that the communication middleware builds on top of L1-L4 of the OSI stack, which forces the middleware implementations to rely on deterministic lower layers for overall end-to-end real-time capabilities. KRS currently provides support for the most popular communication middlewares in the ROS 2 world. Future enhancements are planned to make these implementations rely on real-time deterministic lower layers.

F. ROS 2 Core Layers Indeterminism: Similar to LLC contention, the ROS 2 layers should be built using real-time safe primitives and real-time capable underlayers. KRS initially supports the official ROS 2 `rclcpp`, `rcl`, `rmw`, and `rmw_adapters`. Ongoing work is in progress to explore other architectures that push some of the ROS 2 abstractions to hardware [Ref 11].

G. Robotic Applications and Libraries Indeterminism: At the application level, the robotics code including all its libraries also need to be real-time safe and use only the underlying primitives and layers that guarantee determinism.

Use Cases

Accelerating ROS 2 Applications in C++ with HLS, Building a First Robot Device

Consider a double vector-add ROS 2 publisher that adds two vectors and attempts publishing the result in a selected ROS 2 topic, at a frequency of 10Hz. The vector add operation (VADD) is as well known in the hardware acceleration domain as the *hello world* example. Selected for its simplicity, when combined with ROS 2, it allowed Xilinx to study and describe how a robotics architect could proceed to obtain higher performance and more determinism.

A ROS 2 publisher is presented in code listing 1 in Figure 6, with the function performing the vector add operation at code listing 2. The computations of the VADD are purposely made expensive for the sake of demonstrating acceleration. Of note, computationally expensive vector additions can be common across ROS 2 packages. The source code runs initially in the quad Arm® Cortex®-A53 CPU of the KV260 development board, and is not able to meet the target rate of 10Hz, staying instead at around 2Hz. This value provides us with a computational baseline for the architect to consider when optimizing computations.

After obtaining a baseline, the second step a robotics architect could take to optimize this ROS 2 publisher is to offload the VADD operation from the scalar processors (the CPUs) to specialized robotic circuitry created within the adaptive SoC. In simple terms, create a *robot chip*. By doing so, the architect can leverage the very high potential for parallelism that is exploitable in SoCs to obtain higher throughput, a more resilient security posture, and a strong deterministic response. After all, the VADD operation is implemented in hardware.

Code listing 1 vadd–publisher ROS 2 package host code running in the Processing System (PS), the scalar CPU processors.

```
int main( int argc , char * argv [ ] ) {
  rclcpp : : i n i t ( argc , argv ) ;
  auto node = rclcpp : : Node : : make_shared("
        vadd_publisher") ;
  auto publisher = node–>create_publisher<
        std_msgs : :msg: : String>("vector" , 10) ;
  auto publish_count = 0;
  std_msgs : :msg: : String message;
  rclcpp : : WallRate loop_rate(100ms) ;

  / /  Application variables
  unsigned int in1 [DATA_SIZE] ;
  unsigned int in2 [DATA_SIZE] ;
  unsigned int out [DATA_SIZE] ;

  while (rclcpp : : ok ( ) ) {
    / / randomize the vectors used
    for ( int i = 0 ;  i < DATA_SIZE ;  i++) {
      in1 [ i ] = rand ( ) % DATA_SIZE;   / /  NOLINT
      in2 [ i ] = rand ( ) % DATA_SIZE;   / /  NOLINT
      out [ i ] = 0;
    }

    / / Add vectors
    vadd( in1 , in2 , out , DATA_SIZE) ;   / / function
                                           / /  subject to
                                           / /  be
                                           / /  accelerated
    / /  Validate operation
    check_vadd(in1 , in2 , out) ;

    / /  Publish publish result
    message . data = "vadd finished, iteration: " +
      std : : to_string (publish_count++);
    RCLCPP_INFO(node–>get_logger ( ) , "Publishing:
          '%s'" , message . data . c_str ( ) ) ;

    try {
      publisher–>publish (message) ;
      rclcpp : : spin_some(node) ;
    } catch (const rclcpp : : exceptions : : RCLError & e)
        {
      RCLCPP_ERROR(
        node–>get_logger ( ) ,
        "unexpectedly failed with %s" ,
        e . what ( ) ) ;
    }
    loop_rate . sleep ( ) ;
  }
  rclcpp : : shutdown ( ) ;
  return 0;
}
```

Code listing 2 vadd function of the **vadd–publisher** ROS 2 package. The operation is executed also in the Processing System (PS).

```
void vadd(
        const unsigned int     *in1 ,   / /  Read–Only
                                        / /  Vector  1
        const unsigned int     *in2 ,   / /  Read–Only
                                        / /  Vector  2
        unsigned int *out ,             / /  Output  Result
        int size                        / /  Size in
                                        / /  integer
        )
{
    for ( int j = 0;  j < size ; ++j ) {     / /  stupidly
                                             / /  iterate
                                             / /  to
                                             / /  generate
                                                  load
      for ( int i = 0;  i < size ; ++i ) {
        out [ i ] = in1 [ i ] + in2 [ i ] ;
      }
    }
}
```

WP540_06_091321

Figure 6: **Code Listing 1 and 2**

For this to happen, the ROS 2 publisher needs to be changed by using OpenCL to communicate between the KV260 CPUs and the FPGA, leveraging the Xilinx Runtime tool (XRT), a KRS component that facilitates interactions with the FPGA. Code listing 3 (Figure 7) shows the modifications the architect would need to make to the original ROS 2 package on its kernel source code. Code listing 4 (Figure 8) shows the additions to the `CMakeLists.txt` file for the `colcon` build to build the kernel. KRS automates most of this flow and provides architects a ROS 2-centric experience, allowing them to focus on their individual robotic computation, and solving most of the embedded and hardware details for them.

Code listing 3 Kernel for the accelerated_vadd_publisher ROS 2 package annotated with HLS pragmas.

```
1  void vadd(
2          const unsigned int  *in1,  //  Read-Only Vector 1
3          const unsigned int  *in2,  //  Read-Only Vector 2
4          unsigned int  *out,        //  Output Result
5          int  size                  //  Size in integer
6          )
7  {
8  #pragma HLS INTERFACE m_axi port=in1 bundle=aximm1
9  #pragma HLS INTERFACE m_axi port=in2 bundle=aximm2
10 #pragma HLS INTERFACE m_axi port=out bundle=aximm1
11
12     for  (int  j  +  0;  j  <size;  ++j)  {  //  stupidly iterate over
13                                            //  it  to  generate  load
14 #pragma HLS loop_tripcount min = c_size max = c_size
15     for  (int  i  =  0;  i  <size;  ++i)  {
16 #pragma HLS loop_tripcount min = c_size max = c_size
17           out  [i]  i  =  in1 [i] + in2 [i];
18       }
19   }
20 }
```

The HLS INTERFACE pragma helps specify how RTL ports are created. In here, we have two vectors as inputs that we would like to read simultaneously. We thereby map arguments to ports and request an additional **m_axi** port.

HLS loop_tripcount pragma informs the HLS compiler about how many iterations (in total) the loop will execute. This is then used by the compiler while at synthesis to estimate the overall computation time.

WP540_07_091321

Figure 7: **Code Listing 3: Kernel for the `accelerated_vadd_publisher` ROS 2 Package, Annotated with HLS Pragmas**

Code listing 4 ROS 2 package extension on its **CMakeLists.txt** file to generate a kernel of the type selected.

```
1  vitis_acceleration_kernel(
2    NAME vadd
3    FILE src/vadd.cpp
4    CONFIG src/kv260.cfg
5    INCLUDE
6      include
7    TYPE
8      # sw_emu
9      # hw_emu
10     hw
11   PACKAGE
12 )
```

ament_vitis macro, helps define acceleration kernels directly from CMakeList.txt

kernel name

platform configuration

source code

build (kernel) type

invoke Vitis compiler package flag

WP540_08_091321

Figure 8: **Code Listing 4: ROS 2 Package Extension on `CMakeLists.txt` to Generate a Kernel of the Type Selected**

By default, when creating a kernel, all arguments are assigned to a single AXI interface. While this helps save device resources, it can also limit the performance because all memory transfers between the CPU and the FPGA go through a single port. Moreover, while the `m_axi` port has independent READ and WRITE channels, each channel can only be assigned to one source at the same time. In other words, a single `m_axi` port cannot READ from two sources simultaneously. Since there are two input sources as arguments (in1 and in2), the overall data flow can be optimized by asking for an additional AXI interface and separating the input reads in two different ports.

After introducing these changes and building the kernel, the robotics architect can observe that the ROS 2 publisher is able to publish at approximately 6.2Hz to the computational graph while running on the Kria SOM KV260 board. An initial speedup of more than three times after the small data flow changes introduced in code listing 3 (Figure 7) when compared to the initial baseline, but still below the 10Hz set as the target. The changes introduced only optimize the data flow between the CPU and the FPGA, but no further parallelism in the source code is exploited.

Code listing 5 (Figure 9) shows how to do this. Besides the data flow optimizations between input and output arguments in the PL-PS interaction, line 16 utilizes a pragma to unroll the inner `for` loop by a factor of 16, executing 16 sums in parallel within the same clock cycle. The value of 16 is not arbitrary, but selected specifically to consume the whole bandwidth (512 bits) of the `m_axi` ports at each clock cycle. To fill in 512 bits, 16 unsigned `int` inputs are packed together, each of four bytes (16 unsigned `int`).

Altogether, this leads to the most optimized form of the VADD kernel, which is able to successfully meet the publishing target of 10Hz, obtaining a five times speedup.(1)

Code listing 5 Kernel for the faster_vadd_publisher ROS 2 package modified and annotated with HLS pragmas.

```
1  void vadd(
2              const unsigned int  *in1,  //  Read-Only Vector 1
3              const unsigned int  *in2,  //  Read-Only Vector 2
4              unsigned int  *out,        //  Output Result
5              int  size                  //  Size in integer
6              )
7  {
8  #pragma HLS INTERFACE m_axi port=in1 bundle=aximm1
9  #pragma HLS INTERFACE m_axi port=in2 bundle=aximm2
10 #pragma HLS INTERFACE m_axi port=out bundle=aximm1
11
12       for (int j + 0; j <size; ++j) { // stupdly iterate over
13                                       // it to generate load
14 #pragma HLS loop_tripcount min = c_size max = c_size
15       for (int i = 0; i <size; ++i) {
16 #pragma HLS UNROLL factor=16
17 #pragma HLS loop_tripcount min = c_size max = c_size
18             out [i] i = in1 [i] + in2 [i];
19           }
20      }
21 }
```

The HLS INTERFACE pragma helps specify how RTL ports are created. In here, we have two vectors as inputs that we would like to read simultaneously. We thereby map arguments to ports and request an additional m_axi port.

HLS loop_tripcount pragma informs the HLS compiler about how many iterations (in total) the loop will execute. This is then used by the compiler while at synthesis to estimate the overall computation time.

Unrolls the inner for loop by a factor of 16, executing them all in parallel within the same clock cycle. The value of 16 is not arbirary, but the one that allows to consume the whole bandwidth of the AXI interface on the platform at each clock cycle (512-bit).

WP540_09_091321

Figure 9: **Code Listing 5: Kernel for the faster_vadd_publisher ROS 2 Package Modified and Annotated with HLS Pragmas**

The same principles can be applied to other ROS 2 packages, creating specialized circuitry able to obtain accelerate ROS 2 applications and libraries written in C++ with the help of HLS(2).

Mixed Criticality and Accelerating ROS 2 Underlayers

As indicated above in Real-time at the Core of KRS, real-time is an end-to-end characteristic of a robotic system. For a ROS 2 application to be deterministic and respond in real-time, all the underlaying abstraction layers (ROS 2 underlayers) need to also respond deterministically. KRS provides multiple pre-built components that can be selected directly from the ROS 2 meta build tool (`colcon`), helping architects build more deterministic robotic systems easily. Figure 10 shows two examples of how KRS extends `colcon` (through `colcon_acceleration`) to select a fully

1. In fact, the speedup is higher than five times, however the ROS 2 WallRate instance is set to 100ms, so the kernel is idle waiting for new data to arrive, discarding further acceleration opportunities.

2. With the recent acquisition of Silexica by Xilinx, their technology will be integrated into the Vitis ecosystem. This will significantly reduce necessary code changes in C++ programs, reducing time to market and making HLS more accessible to software developers. The Silexica technology includes automatic identification of parallelism, exhaustive design space exploration for evaluating performance/area trade-offs, and inserting HLS pragmas automatically into the user's source code. See: https://www.xilinx.com/about/xilinx-ventures/silexica.htm

preemtible Linux kernel and a mixed critical setup in the same KV260 board involving the Xen hypervisor with three virtual machines.

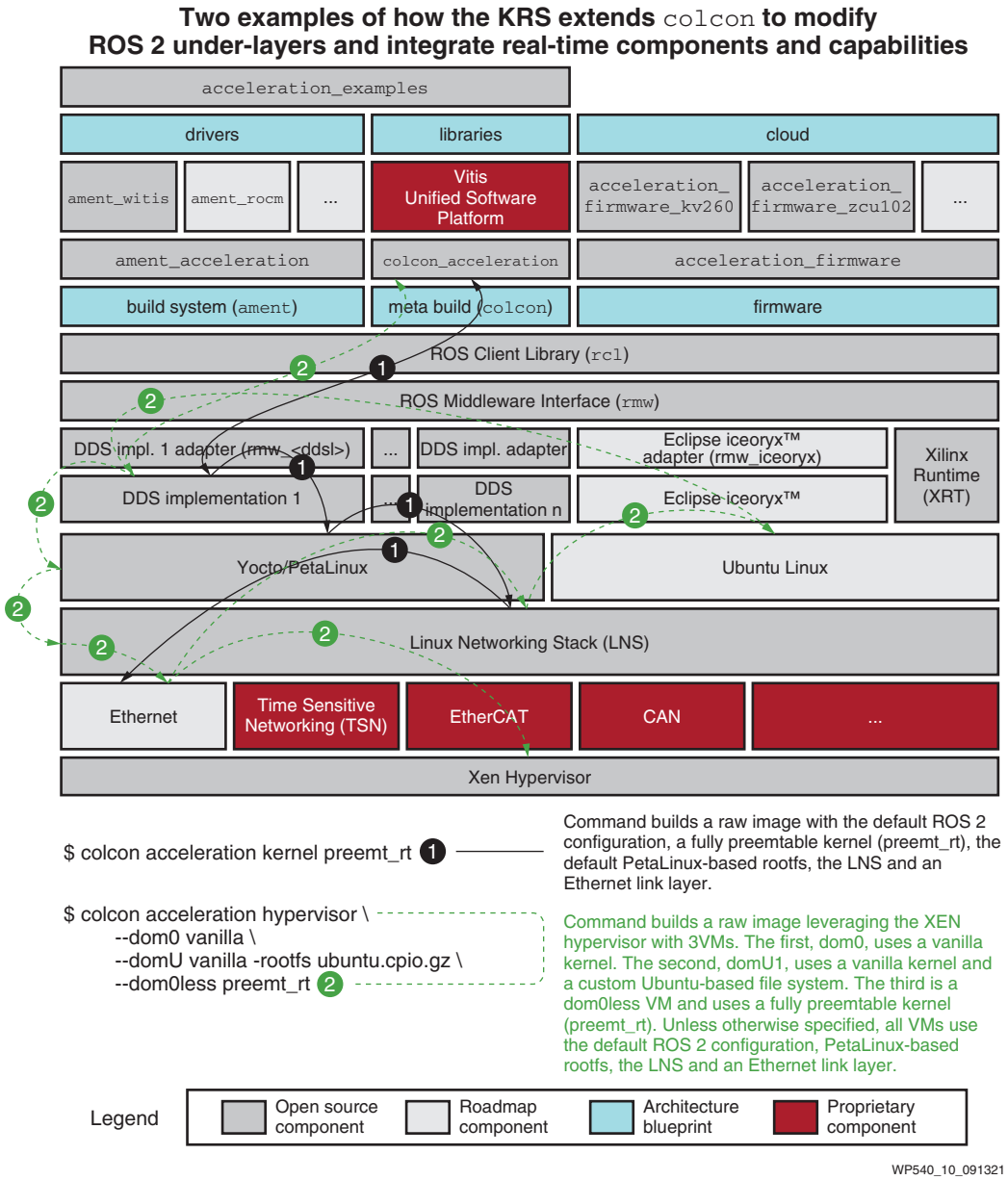


Figure 10: **Examples using `colcon` to Modify ROS 2 Underlayers and Integrate Real-time Components and Capabilities**

Xilinx App Store

The ROS ecosystem is built around a series of abstractions[1] organized in packages. Each ROS 2 package is considered a single unit of robotics software and includes source code, build system files, documentation, tests, quality declarations, libraries, configuration files, and other associated resources that logically constitutes a useful module. Packages provide functionality that is easy-to-consume and reusable by following the Goldilocks principle: enough functionality to be useful, but not too much that the package is heavyweight and difficult to use from other software. By combining different packages, roboticists build computational graphs and with them, robot behaviors.

The process to commercialize a ROS 2 accelerated package for the Xilinx App Store is shown in Figure 11. It depicts the ROS 2 development flow starting with sourcing the local ROS 2 installation, then creating an overlay workspace and building the acceleration kernels, packaging the resulting overlay for commercialization, and making it available in the Xilinx App Store.

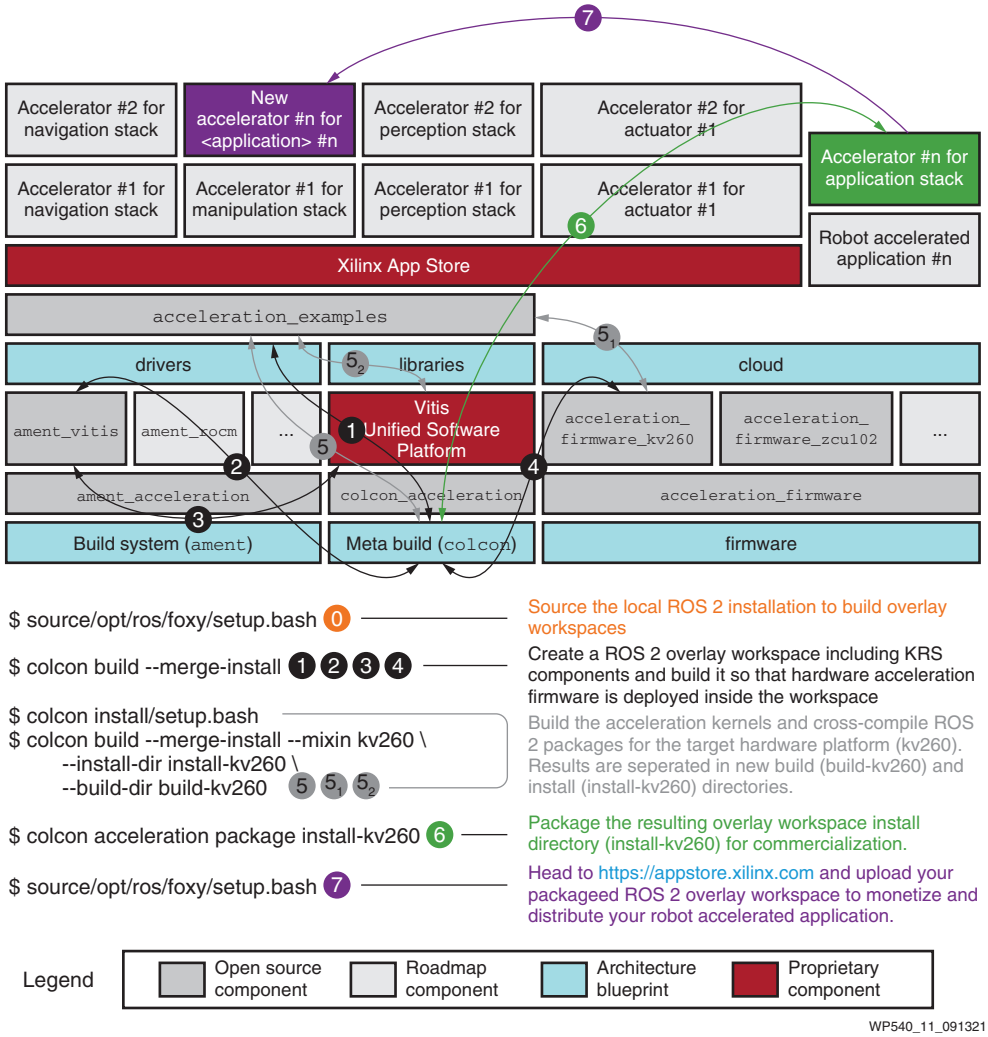


Figure 11: **Process to Commercialize a ROS 2 Accelerated Package in the Kria App Store**

1. Refer to *Adaptive computing in robotics, ROS 2 Software-Defined Hardware* (WP537) for more details on basic ROS 2 concepts.

Packages are packed together in meta-packages, and served in `git` repositories across different sites, most commonly found in GitHub. A subset of all the available ROS 2 packages, each at its own version, is released as a distribution(1). ROS 2 distributions are summarized at https://docs.ros.org/en/galactic/Releases.html. Roboticists usually use ROS 2 distributions as the reference SDK for their robotics development, but often create their own packages inside ROS 2 workspaces, as overlays of the base ROS 2 installation. These ROS 2 overlay workspaces contain the final robotic applications(2).

The concept of ROS 2 packages and overlay workspaces fits nicely with the Xilinx App Store, which offers a powerful platform to host, market, and sell individual acceleration solutions using a managed, easy to use, secure digital rights management (DRM) infrastructure. Discoverable to global customers, the Xilinx App Store is a commercialization hub that allows selected robotics partners to market their inventions in a secure manner, and in the form of ROS 2 overlay workspaces, exactly as in their development environments. Figure 11 depicts the process to commercialize a ROS 2 accelerated package within the Xilinx App Store.

Conclusion

This white paper introduced the Kria Robotics Stack, an integrated set of robot libraries and utilities to accelerate the development, maintenance, and commercialization of industrial-grade robotic solutions. KRS leverages ROS 2 as its robotics SDK and follows a ROS 2-centric approach, building value around its community and other tools, including the Gazebo simulator. KRS specifically targets the Kria SOM hardware portfolio to deliver a production-ready, low latency (fast computation), deterministic (predictable), real-time (on time), secure, and high throughput value to robotics. KRS gives ROS 2 roboticists the capability to create custom secure compute architectures, and it gives selected partners the chance to commercialize these architectures via the Xilinx App Store.

After introducing the different computational models in robotics and describing the design principles behind KRS, this white paper introduced two use cases of KRS. First, adopting the position of a robotics architect optimizing a computation, the white paper shows how to build an acceleration kernel for a trivial ROS 2 application programmed in C++ with HLS. The resulting kernel provides specialized circuitry that allows to obtain a speed up of five times. Second, the white paper demonstrates how to leverage KRS mixed criticality and ROS 2 underlayer optimization capabilities to produce images for the target hardware using `colcon` directly. The white paper concludes by analyzing the ROS 2 workspaces and overlays, and how KRS was built specifically around these concepts to connect Xilinx robotics partners directly with the Xilinx App Store, empowering them with the capability to commercialize their ROS 2 workspaces directly.

Future work involves upgrading KRS with more robot libraries and accelerators, specifically around the three categories: ROS 2 applications and libraries, ROS 2 core upgrades, and ROS 2 underlayers. Similarly, KRS will continue improving its integration with ROS 2 throughout its distributions, build system, and tools. Also, KRS will be releasing additional tooling to facilitate the development experience with Kria SOMs. This includes network booting, additional security mechanisms or benchmarking capabilities for accelerators among others. Get started with KRS at https://xilinx.github.io/KRS.

1. Refer to REP-2005 for more information on the ROS 2 common packages filling ROS 2 distributions.
2. Refer to REP-0128 for more information about ROS workspaces and overlays. The concepts are framed for `catkin`, but equally applicable to `colcon`.

References

1. Z. Wan, B. Yu, T. Y. Li, J. Tang, Y. Zhu, Y. Wang, A. Raychowdhury, and S. Liu, "A survey of FPGA-based robotic computing," IEEE Circuits and Systems Magazine, vol. 21, no. 2, pp. 48-74, 2021.
2. V. Mayoral-Vilches and G. Corradi, "Adaptive computing in robotics, leveraging ROS 2 enabled software-defined hardware," Xilinx, WP537, 2021.
3. M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in ICRA workshop on open source software, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.
4. V. Mayoral-Vilches, A. Hernández, R. Kojcev, I. Muguruza, I. Zamalloa, A. Bilbao, and L. Usategi, "The shift in the robotics paradigm-the hardware robot operating system (h-ros); an infrastructure to create interoperable robot components," in 2017 NASA/ESA Conference on Adaptive Hard- ware and Systems (AHS). IEEE, 2017, pp. 229-236.
5. C. S. V. Gutiérrez, L. U. S. Juan, I. Z. Ugarte, and V. Mayoral-Vilches, "Time-sensitive networking for robotics," arXiv preprint arXiv:1804.07643, 2018.
6. C. S. V. Gutiérrez, L. U. S. Juan, I. Z. Ugarte, and V. Mayoral-Vilches, "Real-time Linux communications: an evaluation of the linux communication stack for real- time robotic applications," arXiv preprint arXiv:1808.10821, 2018.
7. C. S. V. Gutiérrez, L. U. S. Juan, I. Z. Ugarte, and V. Mayoral-Vilches, "Towards a distributed and real-time framework for robots: Evaluation of ROS 2.0 communi- cations for real-time robotic applications," arXiv preprint arXiv:1809.02595, 2018.
8. C. S. V. Gutiérrez, L. U. S. Juan, I. Z. Ugarte, I. M. Goenaga, L. A. Kirschgens, and V. Mayoral-Vilches, "Time synchronization in modular collaborative robots," arXiv preprint arXiv:1809.07295, 2018.
9. Y. Yang and T. Azumi, "Exploring real-time executor on ros 2," in 2020 IEEE International Conference on Embedded Software and Systems (ICESS). IEEE, 2020, pp. 1-8.
10. J. Staschulat, I. Lütkebohle, and R. Lange, "The rclc executor: Domain-specific deterministic scheduling mechanisms for ros applications on microcontrollers: work-in-progress," in 2020 International Conference on Embedded Software (EMSOFT), 2020, pp. 18-19.
11. C. Lienen and M. Platzner, "Design of distributed reconfigurable robotics systems with reconros," 2021.
12. Xilinx, "Xilinx IEC 62443 Compliant Product Enablement," Xilinx, WP513, 2019.
13. Xilinx, "Risk Management for Medical Device Embedded Systems," Xilinx, WP511, 2019.
14. N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on, vol. 3. IEEE, 2004, pp. 2149-2154.
15. R. Casadesus-Masanell and G. Llanes, "Mixed source," Management Science, vol. 57, no. 7, pp. 1212-1230, 2011.
16. M. Bechtel and H. Yun, "Denial-of-service attacks on shared cache in multicore: Analysis and prevention," in 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE, 2019, pp. 357-367.
17. G. Corradi, S. Sabellini, "XEN for Real-Time Interference-Free Virtualization with Zynq® UltraScale+™ MPSoC." 2020. (On-Demand).

Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
09/13/2021	1.0	Initial Xilinx release.

Disclaimer

Automotive Applications Disclaimer