D4.1

Release of the O/S, boot environment, compiler, and performance tools for the FPGA-based single core system
Version 1.0

Document information

Work package:	**WP4: Software Applications Use Cases, Specifications and Evaluation**
Contract number:	956702
Project website:	www.eprocessor.eu
Author(s)	Manolis Marazakis (FORTH)
Contributors	1- BSC: Estanislao Mercadal Melià, Marc Perello, Roger Ferrer 2- CHAL: Nikela Papadopoulou 3- FORTH: Panagiotis Peristerakis, Vassilis Flouris 4- UNIRM: N/A 5- COR: N/A 6- CHR: Stefan Krupop 7- UNIBI: Lennart Tigges 8- EXTOLL: N/A 9- THALES: Jean-Marc Philippe 10- EXAP: N/A
Reviewer(s)	UNIBI (Jens Hagemeyer)
Dissemination Level	PU
Nature	OTHER
Contractual deadline:	31/05/2022 (M14), shifted to 31/07/2022 (M16)

This project has received funding from the European High-Performance Computing Joint Undertaking Joint Undertaking (JU) under grant agreement No 956702. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Sweden, Greece, Italy, France, and Germany.

Change Log

Version	Author(s)	Comments and Description of change
1.0	Editor(s)	Release for formal submission
0.9	Editor(s)	Addressing internal review comments (cont'ed)
0.85	Editor(s)	Consolidation of content sections (incl. Creation of Appendix); Addressing internal review comments
0.8	Reviewer(s)	Internal review comments, misc. edits
0.7	Contributors from BSC, CHAL, CHR, UNIBI, FORTH, THALES	Round-2 of input by deliverable contributors, collection of software artifacts on project file sharing service (B2DROP)
0.4	Contributors from BSC, CHAL, CHR, UNIBI, FORTH, THALES	Round-1 of input by deliverable contributors
0.1	Editor(s)	First release of table-of-contents

Executive Summary

This document provides a summary of work carried out during the first 16 months of the eProcessor project in the context of WP4 , as an accompaniment to the first release of the O/S, boot environment, compiler, and performance tools.

The eProcessor project has created the first version of an FPGA-based single-core system. However, as this work has been performed in parallel to the CPU development, we are not yet covering support for a full Linux OS in this document. Prerequisites for such a full Linux functionality are, e.g., support for atomic operations and virtual memory. Instead, for the first release, we have relied on three alternative platforms to create and validate the main low-level system software artifacts:

(i) QEMU for facilitating tests of alternative Linux kernel configurations for the RISC-V architecture, including configurations with no MMU support;
(ii) an FPGA-based single-core prototype for developing support for RISC-V Linux in an asymmetric multiprocessing configuration, based on the OpenAMP framework and using an older open-source RISC-V core;
(iii) an FPGA-based prototype, coming from related work in the EPI project, for fleshing out bare-metal as well as full-Linux support for a vector-capable RISC-V architecture.

The effort documented in this deliverable provides a solid basis for advanced Linux support on eProcessor prototypes. The bulk of our work so far is expected to work as-is on the upcoming prototypes of eProcessor, with some adjustments in low-level firmware and OS code to handle specific details of the initialization sequence for the processing core. It will also help us address eProcessor-specific aspects of the system, such as the boot cycle and the offloading to accelerators (since those are the functionalities that require special handling - otherwise a stock/vanilla kernel would suffice).

Moreover, this document describes the work toward providing I/O peripherals support for eProcessor prototypes, in the form of a Companion FPGA that provides access to peripherals such as Ethernet,

SATA, USB, and PCIe. The Companion FPGA also serves as the board controller for the upcoming eProcessor microserver module, offering sophisticated power monitoring and management functionality.

Finally, this document provides short updates on work in the following topic areas:

- Compilers and runtimes, regarding work on the LLVM compilation toolchain and the OpenMP threading environment;
- Efficient resource management, via an energy-aware task scheduler operating in a runtime environment based on LLVM and OpenMP;
- Software support for fault tolerance, with a case study on the checkpointing of DNN models;
- Performance monitoring and debug tools, including Extrae for trace capture, the gdb debugger, and DynamoRIO for dynamic binary instrumentation.

Contents

List of Figures

List of Tables

1. Introduction

This document provides a summary of work carried out during the first 16 months of the eProcessor project in the context of WP4 , as an accompaniment of the first release of the O/S, boot environment, compiler, and performance tools.

The eProcessor project has created the first version of an FPGA-based single-core system. However, as this work has been performed in parallel to the CPU development, we are not yet covering support for a full Linux OS in this document. Instead, for the first release, we have relied on three alternative platforms to create and validate the main low-level system software artifacts:

(i) QEMU emulator for facilitating tests of alternative Linux kernel configurations for the RISC-V architecture, including configurations with no MMU support;
(ii) an FPGA-based single-core prototype for developing support for RISC-V Linux in a asymmetric multiprocessing configuration, based on the OpenAMP framework and using an older open-source RISC-V core;
(iii) an FPGA-based prototype coming from related work in the EPI project, for fleshing out bare-metal as well as Linux support for a vector-capable RISC-V architecture.

These platforms and the work carried out on them are described in the remainder of this document. Moreover, this document describes the work towards providing I/O peripherals support for eProcessor prototypes, in the form of a Companion FPGA that provides access to peripherals such Ethernet, SATA, USB, and PCIe. The Companion FPGA also serves as the board controller for the upcoming eProcessor microserver module, offering essential power monitoring and management functionality.

The associated low-level software artifacts for OS support and the I/O functionality (incl. essential configuration files and command scripts for initialization) have been collected on the project's file-sharing service.

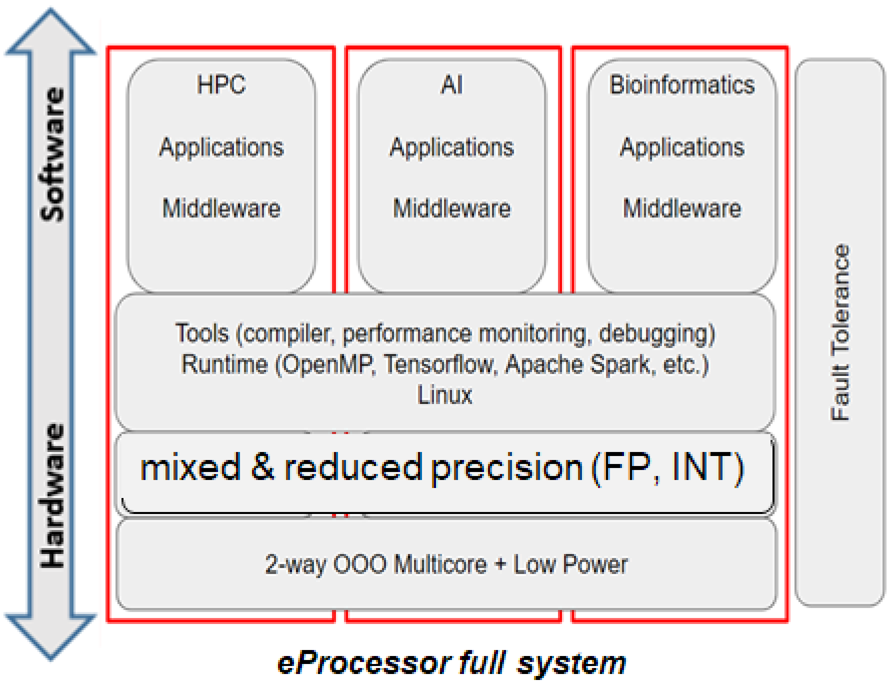

Software
Hardware
HPC
Applications
Middleware
AI
Applications
Middleware
Bioinformatics
Applications
Middleware
Tools (compiler, performance monitoring, debugging)
Runtime (OpenMP, Tensorflow, Apache Spark, etc.)
Linux
mixed & reduced precision (FP, INT)
2-way OOO Multicore + Low Power
Fault Tolerance
eProcessor full system

Figure 1: Overview of eProcessor hardware and software.

Figure 1 provides an outline of the overall hardware and software environment being developed in the eProcessor project. This document summarizes essential aspects of the low-level Linux-based system software, and provides a short introduction to other essential software stack elements: compiler toolchain, runtime systems, support for software-controlled fault tolerance via checkpointing, and debug support, system monitoring, and performance analysis tools.

The effort documented in this deliverable provides a solid basis for advanced Linux support on eProcessor prototypes. The bulk of our work so far is expected to work as-is on the upcoming prototypes of eProcessor, with some adjustments in low-level firmware and OS code to handle specific details of the initialization sequence for the processing core. It will also help us address eProcessor-specific aspects of the system, such as the boot cycle and the offloading to accelerators (since those are the functionalities that require special handling and otherwise a stock/vanilla kernel would suffice).

2. Operating system support

We have worked with various low-level system software options across a series of platforms with RISC-V (64-bit) architectures to utilize the hardware that is not able to support fully operational Linux. We introduce three platforms of increasing complexity, and utilize them with various software environments, including bare-metal, RTOS, and Asymmetric Multiprocessing.

2.1 Platforms

For the development of a boot and system software stack for the eProcessor system, three platforms are utilized: .

- QEMU: an open source emulator (with RISC-V support), which uses KVM for paravirtualization and is able to run a variety of operating systems.
- Ariane SDV: We use the term Ariane-SDV (software development vehicle) to describe a Trenz Electronic FPGA development board[1] with the open-source Ariane RISC-V processor on FPGA[2]. On the PS (Processing System) side of the development board, Ariane-SDV runs Linux on the ARM Cortex-A53 multicore processor, which is used for management, monitoring and control interface purposes. On the PL (Programmable Logic) side, where an Ariane (riscv64) core is programmed, different modes of operation (will be explained below) can be run. In order to run binaries on Ariane-SDV we have two options:
    - The rvinit script (requires access to the Linux kernel's /dev/mem facility for direct read/write access to the entire memory space available to the kernel) that puts the RISC-V SoC on reset, writes the Zephyr binary in DRAM, and then sets the core's first jump address before finally taking the SoC out of reset.
    - The Ariane remoteproc driver (based on the Xilinx zynqmp_r5 driver) which does the same steps as rvinit, loading an ELF instead of a binary image, which may contain a resource table in order to be compatible with OpenAMP.
- EPI-SDV: We use the term EPI-SDV (European Processor Initiative software development vehicle) to describe an x86 host which is connected, through PCIe, to an FPGA emulating the EPI project's CPU (with the Avispado RISC-V core and other related IP blocks from the EPI project that make-up the EPAC accelerator system) as seen in Figure 2. In order to load binaries on EPI-SDV, we run a script that first resets the FPGA by writing two registers over PCIe, then writes the binary to

[1]https://shop.trenz-electronic.de/en/TE0802-02-2AEV2-A-MPSoC-Development-Board-with-Xilinx-Zynq-UltraScale-ZU2-and-LPDDR4

[2] https://github.com/lowRISC/ariane

DRAM (starting at the base address defined by the platform's linker script), and finally initializes the EPAC environment, by writing to several configuration space addresses over SPI.

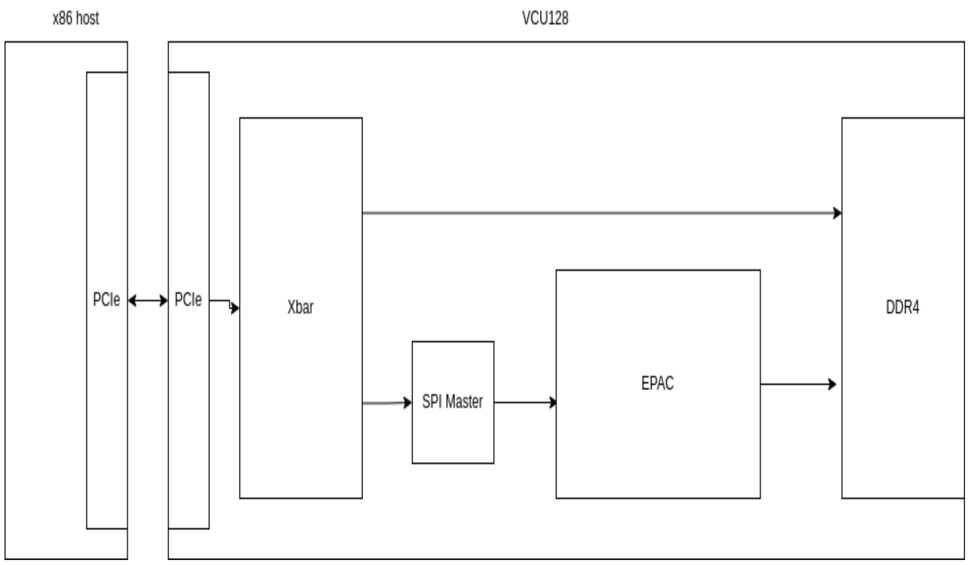


Figure 2: EPI-SDV: connection between Host (x86) and the EPAC RISC-V, over PCIe.

2.2 Modes of operation

We have demonstrated four different modes of operation, each highlighting different aspects of prototype capabilities: Bare-metal code execution, Linux-nommu for a bare-bones Linux kernel environment without virtual memory, Zephyr RTOS for a task-based execution environment without virtual memory support, and Asymmetric Multiprocessing using the OpenAMP framework with coordination and data exchange between a Linux host and a companion processing element. In more detail, the modes of operation have the following characteristics:

- Bare-metal: Bare metal programming is a low-level type of programming that is hardcoded to a system at the hardware level and operates without an abstraction layer or operating system (OS). It also interlinks with the hardware, considering the specific build of components.
- Bare-metal on EPI-SDV: We need to change the rom base address on linker script and to assign the 64 bit uart base address directly to the variable (uart_base) in common/uart.c, because the linker script does not support 64 bit addresses.
- Linux-nommu: This is a configuration for running linux without virtual memory, in a single address space (kernel and applications). Linux runs on M-mode without supervisor (e.g opensbi) and it can be used by CPUs that lack MMU support. In order to run nommu linux on our system, we based our work on qemu riscv64 support, which uses the Uclibc compiler and runs busybox (with position independent code).
- Linux-nommu on QEMU riscv64: In order to run linux nommu on qemu riscv64, one has to take the following steps:
    - Use the custom yaafrv build automation tool to generate a RISC-V Linux binary image (with embedded initramfs).
    - Then: cd to buildroot directory and change branch to "next", which contains qemu_riscv64 nommu configuration.
    - After that, to run "make" command to build the Image and finally run the following command: "qemu-system-riscv64 -nographic -machine virt -cpu rv64,x-v=true,vlen=128,elen=64,vext_spec=v0.7.1 -smp 1 -m 2G -kernel Image -bios Image -append "root=/dev/vda ro" -initrd rootfs.cpio"(see Figure P3). It should also be noted that the BIOSneeds also to be the Image (instead of opensbi), because linux runs on M-mode. In Figure P4, we see the execution of commands on nommu linux.

- Linux-nommu on EPI-SDV: In order to run linux nommu on EPI-SDV, we had to take the following steps.
    - Use yaafrv, an automated flow created by FORTH for creating RISC-V Linux binary images, to generate a bootable Linux binary image (with embedded initramfs).
        - Then: cd to buildroot directory and change branch to "next", which contains qemu_riscv64 nommu configuration.
        - Copy ./configs/qemu_riscv64_nommu_virt_defconfig to .config, in order to use it. Run make menuconfig(buildroot configuration) and enable initramfs(BR2_TARGET_INITRAMFS).
    - The address map of the EPI-SDV is such that 64 bit addresses are required. In particular to set the correct base address for the kernel, the existing default value for the PAGE_OFFSET configuration in the arch/riscv/Kconfig had to be altered:

      From: config PAGE_OFFSET -> default 0x80000000 if 64BIT && !MMU

      To: config PAGE_OFFSET -> default 0x800000400000 if 64BIT && !MMU
    - In order to run linux nommu on EPI-SDV, you need the linux kernel Image (without a bootloader or supervisor), the device tree (which you can produce with qemu command, e.g., qemu-system-riscv64 -nographic -machine virt -cpu rv64 -machine dumpdtb=virt.dtb) and a brom that initializes the hardware (and informs the linux of the device tree blob address) before the linux execution begins. We have to perform some minor modifications on the device tree, including changes in UART block, memory block and CPU frequency.
    - In order to load linux nommu on EPI-SDV, we then need to execute the EPI-SDV boot script mentioned above.
- Zephyr on qemu riscv64: Zephyr is a small real-time operating system (RTOS) for connected, resource-constrained and embedded devices (with an emphasis on microcontrollers) supporting multiple architectures and released under the Apache License 2.0.
    - Zephyr includes a kernel, and all components and libraries, device drivers, protocol stacks, file systems, and firmware updates, needed to develop full application software. The Zephyr build system compiles and links all components of an application into a single application image that can be run on simulated hardware or real hardware.
    - The RISCV64 QEMU board configuration is used to emulate the riscv64 architecture, which we also used as an example in order to run zephyr on other riscv64 systems (Ariane-SDV, EPI-SDV). More instructions on how to build and run qemu on this platform can be found viA the link below:
- Zephyr on Ariane-SDV: We run Zephyr on EPI-SDV, by the following steps:
    - Clone the zephyr github repository.
    - Then use the qemu-riscv_64 board configuration (zephyr/boards/riscv/qemu_riscv64) as a starting point, by copying and changing the directory and file names (we renamed our board as ariane). Our modifications focused mainly on the device tree (ariane.dts), which is needed for building the executables (ELF and binary).
        - One of the main differences between the Ariane and qemu_riscv64 device trees is that Ariane uses 64-bit register addresses (#address-cells and #size-cells =2).
        - The UART has different frequency and clock-speed, as well as registers and reg-shift(2).
        - The Ariane device tree has one core in cpu block, where qemu has 8.
    - Moreover, in the "defconfig" configuration file we had to disable PMP (CONFIG_RISCV_PMP), because it does not work on Ariane. To produce a binary, we had to enable CONFIG_BUILD_OUTPUT_BIN, else only ELF is built.
- OpenAMP on Ariane-SDV:
    - OpenAMP is a framework that standardizes communication between Operating systems(in particular between Linux and RTOS/bare-metal or zephyr). The purpose is to use OpenAMP in order to control accelerators in FPGA, with the Linux side being the master that will initiate the experiments.
    - We used the "OpenAMP Sample Application using resource table" zephyr application (with some modifications) for the Ariane (OpenAMP slave-side) and "rpmsg-sample-ping"

  - (the Linux master-side which was also modified in order to respond to the zephyr application).
  - For the Linux OpenAMP application as our platform we used the zynqmp machine, with some modifications as detailed in the following.
    - In order to use OpenAMP, we added some blocks to the device tree that linux uses, according to the OpenAMP manual. In our case, the remote processor is not R5, but Ariane and we are not using either the zynqmp-ipi-mailbox or tcm memory, therefore we had to make some modifications as well(we need the vrings, vdevbuffer, rproc and a device that probes Ariane remoteproc driver).
    - Each side (Zephyr and Linux) is using polling (instead of mailboxes that the vanilla code uses) in order to notify the other that a message has been sent. First we use the Ariane remoteproc driver which parses the elf, retrieves the information required for OpenAMP (e.g., vrings, vdevbuffer) in order to include them to the resource table and then start the remote processor (Ariane).
    - After that, we start the OpenAMP user space application on Linux and both sides begin to exchange rpmsg messages (rpmsg is a component to OpenAMP for sending messages to remote processors).
    - Due to the way memory addressing works between ARM Cortex A53 and Ariane, we had to add an offset (0x800000000000) on the shared memory components (vrings, vdevbuffer) at the side of Zephyr. The source and destination addresses of rpmsg endpoint need to be defined on both sides, as well as the resource table address (inside the Zephyr ELF) and the physical address in DRAM where the ELF will be loaded should be defined in the Linux user-space code.
- OpenAMP on EPI-SDV: We run OpenAMP between a x86 Linux host and Zephyr running on the EPI-SDV RISC-V core, similarly to the OpenAMP experiment on Ariane-SDV.
  - We currently cannot load ELF executables on EPI-SDV; therefore, we fallback to loading a Zephyr binary, which however cannot contain the resource table of OpenAMP. We had to adjust the “OpenAMP Sample Application using resource table” Zephyr application in order to define all the required memory locations (esp. vrings, vdevbuffer).
  - On the x86 Linux side, we are also using a modified version of “rpmsg-sample-ping” to respond to the Zephyr application (as we did with Ariane-SDV). We have used the Linux generic machine as our platform, with some modifications in order to use PCIe for memory accesses.

The Appendix includes a number of screenshots from the experimental platforms, illustrating the modes of operation outlined above.

2.3 Companion FPGA driver porting

Besides the peripherals integrated into the eProcessor chip itself, there will be an FPGA on the microserver module named the “Companion FPGA”. This provides additional peripherals like, e.g., 1G Ethernet, USB, SATA, and PCIe. The FPGA selected is a Zynq Ultrascale+SoC that, besides the FPGA fabric (Programmable Logic, PL), also contains an ARM-based “Processing System” (PS) with lots of commonly used peripherals implemented as hard blocks. While some of the interfaces used for the eProcessor microserver module will be implemented in the PL (e.g., PCIe and 10G Ethernet), most of the others will be implemented using the hard blocks in the PS. As these peripherals are originally designed for use by the ARM cores in the processing system rather than by an external processor, it was unclear whether utilizing them from the eProcessor would work without major changes. Therefore, a test system using an external processor implemented in the PL was set up, see Figure 3.

The interrupts of the peripherals are typically connected to an internal interrupt controller accessible by the ARM processor. However, the interrupt lines can also be connected to the FPGA fabric (Programmable Logic, PL) via an "Export function" in Xilinx Vivado, and can be connected to an interrupt controller implemented in the PL and accessible by the eProcessor.

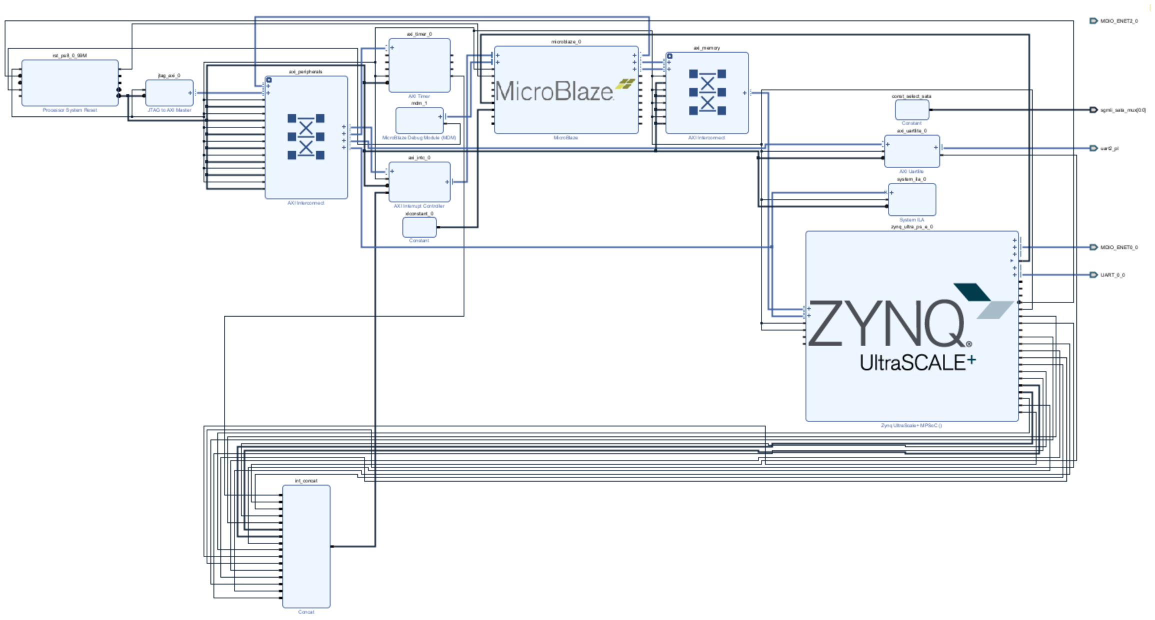


Figure 3: Block design of the test system for the companion FPGA, implemented in Xilinx Vivado.

After being able to boot Linux on that processor (which required some fixes on its own), the peripherals to be tested were added to the Device Tree one by one. For most drivers, it was sufficient to modify the "Kconfig" configuration to allow building them on non-Xilinx platforms. However, some drivers depend on a "Xilinx Zynq MPSoC firmware interface" to be able to control some reset lines, clock- and power domains. This driver needs to communicate with the Platform Management Unit (PMU), a small co-processor included in the MPSoC, as the PMU Firmware running there has exclusive access to these signals. Normally, this is done by calling the "ARM Trusted Firmware" running on the ARM processor via an SMC (Secure Monitor Call) or HVC (Hypervisor Call) instruction. The Trusted Firmware then either interprets the call by itself or forwards it to the PMU Firmware using the IPI (Inter Processor Interrupt) Mailbox built into the SoC, see Figure 4.

As the SMC or HVC mechanisms as well as the ARM Trusted Firmware are not available for the eProcessor, the Xilinx firmware driver was modified to also be able to directly access the IPI Mailbox via register writes that are normally done in the Trusted Firmware. As only the communication mechanism was extended, some parts of the code normally running in the Trusted Firmware environment needed to be ported into the driver so that these calls (e.g., getting version information) also succeeded. The peripherals listed in Table 1 have been tested with the modified firmware driver.

The Linux Kernel repository with the changes to the drivers so far tested can be found at https://gitlab.bsc.es/eprocessor/linux in the "microblaze_peripheral_test" branch.

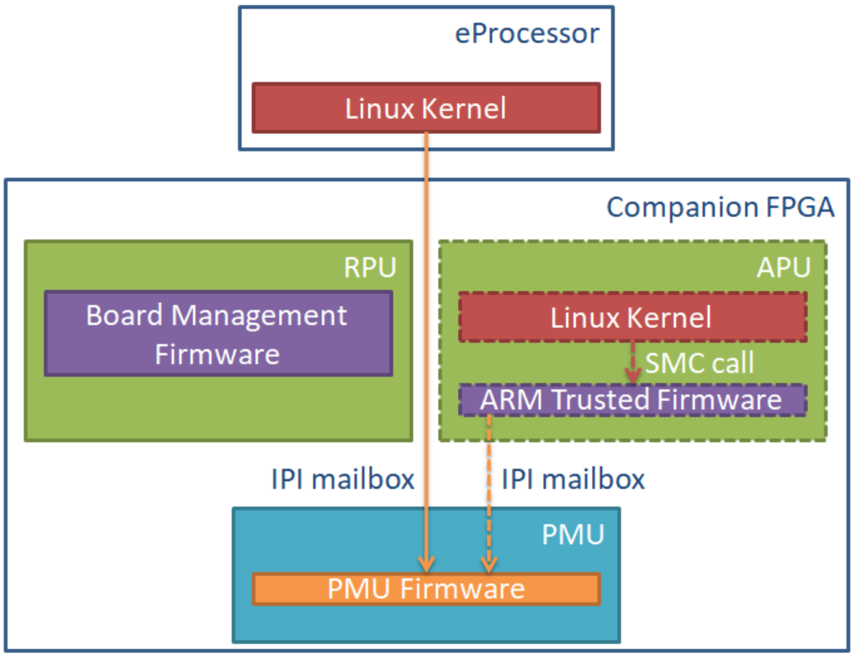


Figure 4: PMU Firmware communication. Dashed box: ARM use case, Solid box: eProcessor case.

Table 1: PS peripherals tested for access from external Processor.

Peripheral	**Status**
DDR (PS)	OK
Low speed IFs (UART, I2C, SPI, GPIO)	OK
1 GB Ethernet	OK
SD-Card / eMMC	OK
USB 2/3	Mostly working
SATA	Peripheral accessible, no SATA link yet due to 3rd party issue
GPU / DisplayPort	Pending

3. Operating system support

Work in WP4 includes the development of resource management with energy awareness, for multithreaded workloads using the OpenMP runtime environment.

3.1 Extensions in ERASE (EneRgy Aware SchedulEr)

ERASE[3] is an energy-aware task scheduler, developed by Chalmers, and implemented in the XiTAO runtime[4]. ERASE relies on power profiling, performance modeling and core activity tracing to achieve energy-efficient mapping of tasks, improving the energy consumption of task-based applications. The initial implementation of ERASE in XiTAO is based on (1) online history-based prediction of the execution time of tasks, and (2) offline power profiling on different core types, and numbers of cores.

We are currently working towards extending the model building component on top of ERASE. In particular, we are implementing performance models, as well as power models for the CPU and the memory components, which are able to predict performance and power, given more dimensions as input, namely the CPU core type, numbers of cores, CPU frequency and memory frequency. Models with higher predictive ability will be able to improve the energy efficiency decisions of the scheduler. We have validated our models on execution scenarios where resources are not shared between tasks. As a next step, we will develop models which take into account resource sharing (e.g. sharing memory).

3.2 ERASE in LLVM-OpenMP

As a step to achieve energy efficiency in widely adopted runtimes, we have implemented ERASE in OpenMP, focusing on *kmp*[5], the OpenMP implementation in LLVM. The techniques developed in ERASE focus on task-based applications. The LLVM OpenMP runtime implements the tasking model with a set of local queues, one per thread, offering lower scheduling overheads than the equivalent implementation in libgomp[6]. Additionally, the design concept of local queues is a better fit for the implementation of ERASE, which has originally been developed within the XiTAO runtime.

The first prototype of ERASE in LLVM OpenMP implements the basic concepts of ERASE. The goal of ERASE is to achieve the reduction of the energy consumption of task-based applications. To achieve this, ERASE relies on predictive models. In detail, ERASE monitors and predicts the execution time of each task online. In the original implementation of ERASE in XiTAO, performance modeling is history-based. In the OpenMP implementation of ERASE, we additionally support online monitoring of the performance characteristics of the various tasks using *perf*[7], and combine the collected metrics into interpolation models that comply with PMNF[8]. Models can be built either online or offline, with the latter showing better accuracy. The two main techniques in ERASE leverage predictive modeling, as follows:

- *Task moldability*. In LLVM OpenMP, task moldability is implemented with the *taskloop* work-sharing construct in OpenMP. The *taskloop* construct splits the iterations of a loop into individual tasks. To achieve energy efficiency, the scheduler, encountering the taskloop pragma, determines how many tasks the loop will be split into, depending on the core cluster and core width, i.e. the execution place.
- *Task-type aware execution.* In combination with task moldability, ERASE examines task properties to determine the optimal execution place for each task. Using monitoring and

[3] Chen, Jing, Madhavan Manivannan, Mustafa Abduljabbar, and Miquel Pericas. "ERASE: Energy efficient task mapping and resource management for work stealing runtimes." *ACM Transactions on Architecture and Code Optimization (TACO)* 19, no. 2 (2022): 1-29.

[4] XiTAO Runtime [Online]. Available: https://github.com/CHART-Team/xitao

[5] LLVM, "Openmp* runtime library," 2015. [Online]. Available: https://openmp.llvm.org/

[6] GNU libgomp [Online]. Available: https://gcc.gnu.org/onlinedocs/libgomp/

[7] Linux perf [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[8] Calotoiu, Alexandru, David Beckinsale, Christopher W. Earl, Torsten Hoefler, Ian Karlin, Martin Schulz, and Felix Wolf. "Fast multi-parameter performance modeling." In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 172-181. IEEE, 2016.

predictive modeling, the scheduler creates and fills a performance table for each task type, to make informed decisions on task placement.

The implementation can be found at: https://b2drop.bsc.es/index.php/s/mccEy6oSw6cE5fL. We have successfully compiled and executed this version of the runtime on the eProcessor QEMU image, noting, however, that only the version of the runtime that relies on the history-based online performance modeling is available, due to the absence of *perf*.

We have extensively tested the implementation of ERASE in OpenMP on NVIDIA JETSON TX2 (an Arm-based development system), where we have found ERASE-OpenMP to achieve an average of 7% reduction in energy consumption across benchmarks of the PARSEC and BOTS suites, with a maximum energy reduction of 15%.

Next steps regarding the implementation of ERASE in OpenMP include:

- Optimizing the current implementation, in terms of instruction and memory footprint;
- Improving the modeling methodology, to achieve accurate performance modeling with online model creation;
- Validating the functionality of ERASE-OpenMP on eProcessor SDVs.

Our initial evaluation of ERASE on alternative platforms has demonstrated that it can achieve a reduction of more than 30% in the energy consumption of task-parallel applications. ERASE targets the final multicore eProcessor platform, where we expect that we will be able to achieve similar and higher energy savings, exploiting the eProcessor platform's knobs for frequency and cache management.

4. Software support for fault tolerance

Work in WP4 includes the development of software-based fault-tolerance techniques, based on application-directed checkpointing. This section provides an overview of a multi-level checkpointing library for HPC, and a case study focused on large-scale DNN training.

4.1 Optimizing the checkpointing of DNN models

Deep learning is an emerging machine learning technique that has had impressive results when tackling problems in a variety of fields, such as medical image analysis, computational biology and natural language processing. Powerful DL models have been developed in order to solve complex problems. However, these models tend to have a large amount of parameters. A light-weight model such as MobileNet, which has been optimized to run in low power devices, has over 4 million parameters. The widely known ResNet50 model, used a lot in academia, has over 25 million parameters. However, for very complex problems, these models might not produce good enough results. Google’s GPT is an extremely powerful DL model for unsupervised learning, and has a total of 175 billion parameters.

The training of large models is a very time-consuming task, taking from hours to weeks in a single machine. Because of this, large-scale training approaches have been developed, distributing the training work across several machines. This has put DNN training in close relation with HPC environments. Fault tolerance is crucial in HPC environments, where large quantities of components are used, which harshly reduces the overall MTBF of the system. Checkpoint-Restart is a widely known fault tolerance technique used in HPC. Moreover, it is also used in DNN training not only for fault tolerance, but also for optimization techniques such as transfer learning and hyper-parameter tuning. As such, it makes sense that large-scale DNN training in HPC environments would also use these checkpointing techniques. While common DNN frameworks[9,10] provide the ability to checkpoint, it was not developed with large-scale operation in mind. As such, they simply perform a checkpoint of the entire model in a single storage medium. On the other hand, state-of-the-art checkpoint libraries in HPC[11,12],[13] are not optimized for performing checkpoints of DNN models. This is because in most cases the DNN model data is synchronized across all training processes before checkpointing, which causes all checkpoints to be identical. As such, performing the entire checkpoint in all processes is very inefficient and wasteful.

In order to obtain the benefits of HPC checkpointing while also handling DNN models efficiently, we developed PyFTI. PyFTI is a python module that works over FTI[13], a multi-level checkpointing library for HPC. While PyFTI works as a python binding module for FTI, it also provides optimizations for checkpointing DNN models. It currently supports Tensorflow and PyTorch frameworks. PyFTI takes advantage of the fact that DNN model data is identical at the time of checkpoint, and splits this data across processes. This technique is especially beneficial when performing local checkpoints. Since every individual machine has its own local storage, this technique leverages the bandwidth of each local storage device. We can observe an example of this in Figure 5. In this example, the total amount of model data to be checkpointed is 1 GByte, and the training is distributed across 8 nodes. With the standard checkpoint procedure of most DNN frameworks, a single process writes the entirety of the model to storage. On the other hand, with a partitioned strategy, every node only performs a checkpoint of a portion of the DNN model, significantly increasing the overall I/O bandwidth of the system. It is important to note that this comes at the cost of additional communication in the case of recovery, although its overhead is minor.

[9] Tensorflow [Online]. Available: https://www.tensorflow.org/

[10] PyTorch [Online]. Available: https://pytorch.org/

[11] A. Moody, G. Bronevetsky, K. Mohror and B. R. d. Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, 2010, pp. 1-11, doi: 10.1109/SC.2010.18.

[12] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror and F. Cappello, "VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale," 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2019, pp. 911-920, doi: 10.1109/IPDPS.2019.00099.

[13] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama and S. Matsuoka, "FTI: High performance Fault Tolerance Interface for hybrid systems," SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011, pp. 1-12, doi: 10.1145/2063384.2063427.

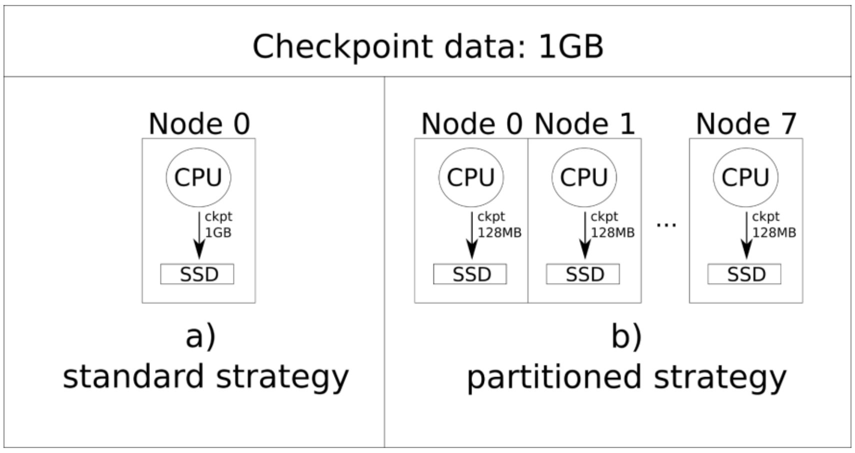


Figure 5: Example of execution with different checkpoint strategies on local storage.

4.2 Saving modes in PyFTI

We have developed different modes in which to checkpointing data. Those techniques can be grouped into 3 types: non-partitioning, raw-partitioning and list-partitioning.

- **Non-partitioning**: as the name implies, those techniques do not split the checkpoint data between processes. This is the standard behavior when the data cannot be assumed to be the same across all processes.
- **Raw-partitioning**: the data is partitioned at a byte level, after serialization. This has the advantage of distributing the checkpoint data equally between processes, but has the disadvantage of having to perform the serialization of the entire object on each process.
- **List-partitioning**: the data is partitioned at an index level. That is, each process extracts a certain number of elements of the list and serializes them individually. This reduces the serialization cost, as each process only serializes the data that they need to save. However, properly distributing the data between processes may prove difficult if the list has elements with large size differences, and can end up with imbalance. In order to alleviate this imbalance, two different selection techniques have been developed:
    - **Fast**: each process gathers a specific number of elements purely based on the index count. This is ideal when the elements in the list have a similar size.
    - **Weighted**: the number of elements is selected taking in consideration an estimation of the size of each element. This selection technique has slightly more computation overhead, but substantially reduces the size imbalance of the checkpoint data.

In the Appendix, we present an evaluation case study comparing these saving modes offered by PyFTI.

4.3 Follow-up work on PyFTI

We are currently working on developing a flexible testbench that allows us to test new PyFTI features and perform exhaustive performance evaluation. This testbench currently allows us to verify the correctness of all Tensorflow and PyTorch modes, along with comparing them with executions without checkpointing and with standard checkpoints from the respective framework. It also allows us to specify which dataset, model and optimizer we want to use, along with other options such as asynchronous checkpointing. We have recently added support for Imagenet dataset in both frameworks, and we are soon going to perform a thorough performance evaluation with all the features that we have implemented so far.

5. Compilers and Runtimes

In this section, we outline work carried out in WP4 towards compiler support (with the LLVM toolchain), computation offloading, and OpenMP runtime optimization.

5.1 LLVM-based Compiler

[Repository link: https://gitlab.bsc.es/eprocessor/eprocessor-llvm]

The LLVM-based compiler developed in the EPI SGA1 project, can be used in the platform of eProcessor. A key feature of the compiler toolchain from the EPI SGA1 project is that it supports the generation of RISC-V vector instructions, including via intrinsics (currently adhering to v.0.7.1 of the V-ISA standard from RISC-V International). However, eProcessor has additional requirements not covered in the EPI project. Those include the ability to execute vector computation with small element sizes: these small element sizes include integers of 4 and 2 bits, and floating point types of 16-bits (IEEE 754 Binary16 and Google's bfloat data types) and 8-bits. The 8-bit floating types are non-standard but in internal discussion with the hardware implementers, two formats (with the same structure as the IEEE 754 binary formats of sign-exponent-mantissa) have been chosen: 1-3-4 and 1-5-4.

In order to have the compiler ready along with the hardware implemented in eProcessor, we have been extending the EPI SGA1 compiler for the RISC-V Vector Extension (RVV) 0.7.1. Support for IEEE 754 Binary16 is already available in the compiler. Now the work is focusing on implementing bfloat. To accommodate the alternate 16-bit format of bfloat with RVV, we have been in contact with the hardware implementers, who have agreed to include an extra bit in the vtype (vector type) register of RVV to designate the alternative format. Assembly and code generation support will use this bit for bfloat and the 1-5-4 format.

5.2 Computation offloading for off-chip CNN accelerator

One of the major results of the eProcessor project will be the ability of the computing infrastructure (hardware and software) to efficiently offload CNN (Convolutional Neural Network) computations to off-chip accelerators. This is an important feature since this computing paradigm is heavily found in HPC systems. In the eProcessor project, the surveillance border control use case aims to illustrate this idea by enabling the hybrid execution of an application on the eProcessor RISC-V processor and an existing off-chip CNN (Convolutional Neural Network) accelerator implemented on an FPGA board. Figure 6 shows the eProcessor chip linked with an FPGA board thanks to a chip-to-chip link and also the partitions between the different software flows.

The CNN application will be partitioned into hardware-accelerated parts (implemented using Keras/Tensorflow frameworks and a dedicated firmware generator) which will be computed on the off-chip accelerator and software-parallelized (via OpenMP) parts which will run on the eProcessor chip. Firmware management, communications and synchronizations between the host processor and the CNN-off-chip accelerator are currently done using a custom API (Application Programming Interface). The efficient exploitation of the off-chip accelerator requires modifications of this API, especially since the link between the eProcessor chip and the off-chip accelerator will use a memory coherent infrastructure, as depicted in Figure 6. The communications between the host and the off-chip accelerator are meant to be simplified since data exchanges will rely on the internal coherency infrastructure of the eProcessor chip and a specific coherency infrastructure on the FPGA board side that will be designed in WP6 during the next period of the project.

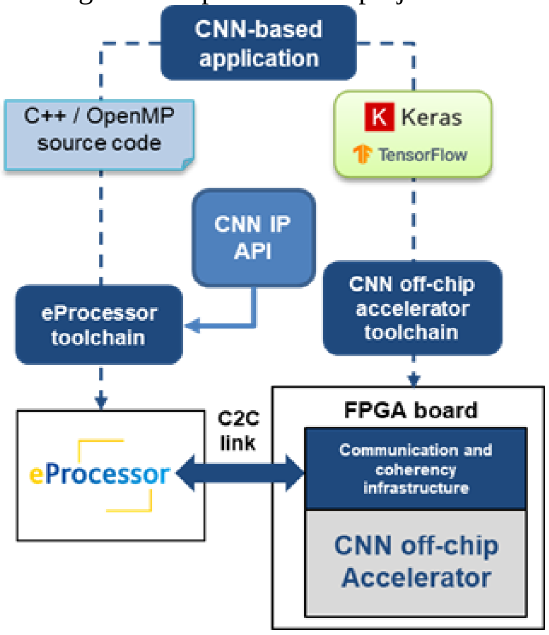


Figure 6: Overview of the hybrid accelerated execution of the CNN application (surveillance border control).

Plans for this FPGA-based infrastructure were documented in deliverable D3.1. At the time of writing this deliverable, deeper insights on the existing APIs regarding the off-chip CNN accelerator were gathered. Once the work on the coherency infrastructure will be sufficiently advanced, real implementations will start. More details are to be provided in upcoming runtime-focused deliverables, namely D4.3 and D4.4.

5.3 Dead block identification in the OpenMP runtime

Research has shown that last level caches (LLCs) are typically underutilized because a large portion of the blocks are dead (i.e. will not be used before eviction). In the eProcessor project CHALMERS is working on developing software and hardware mechanisms that permit efficient utilization of the LLC by identifying and evicting dead blocks early thereby improving cache utilization. More specifically, this involves implementing a static/dynamic dead-block management (DBM) technique that can accurately identify dead blocks, communicate information about such blocks to the LLC and accommodate mechanisms in hardware that can facilitate eviction of such blocks from the LLC. We estimate[14] that DBM can lead to an average reduction in LLC misses of more than 23% when compared to a baseline system without dead block management. Note that in the context of this project dead-block identification and eviction will be implemented at the granularity of address regions, as supported in many of today's task-parallel programming models like OpenMP. In this document we describe the software mechanisms we have incorporated in the LLVM OpenMP runtime system to enable detection of dead regions. Details regarding appropriate hardware-software interfaces that will be used to communicate this information to the LLC and how this information is leveraged to facilitate eviction of dead blocks are presented as part of a separate work package (WP5, T5.3).

In order to understand how we extend the LLVM OpenMP runtime system with support for detecting dead regions we first need to provide an overview about how the runtime system utilizes dependency

[14] M. Manivannan, V. Papaefstathiou, M. Pericas and P. Stenstrom, "RADAR: Runtime-assisted dead region management for last-level caches," 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2016, pp. 644-656, doi: 10.1109/HPCA.2016.7446101.

annotations (address regions accessed by a task marked using in and out depends clause) to establish inter-task dependencies and construct a task directed acyclic graph (DAG). The runtime system establishes dependencies among sibling tasks by letting a master thread first generate tasks in program order. Tasks which do not have any dependencies are released to the ready queue for execution in an out-of-order manner while tasks which have unresolved dependencies are held until their dependencies are resolved. Worker threads pick up tasks from the ready queue for execution. As part of the task initialization process the runtime instantiates a dependency node (DAG node) corresponding to each task. For each of the regions accessed by the task, an entry is created in the dependency hash data structure which tracks the other dependency nodes that read and/or write to this specific address region (in case an entry for the region does not already exist) . In case a dependency entry for an address region already exists, the dependency node representing the task that is currently being instantiated is marked as a successor for the other dependency node (task) that will update the specific address region. In addition a reference counting mechanism, is used by successors to identify the number of predecessors it depends on. After each task finishes execution, the predecessor count for each of its successors is decremented by one. Once the predecessor count reaches zero the successor dependency node can be deemed to not have any unresolved dependencies and is released to the ready queues for execution.

In order to enable dead block detection in the runtime system we incorporate two simple changes to the dependency tracking mechanism used in the runtime system. Firstly we make changes to the dependency node data structure by adding additional information about the region base address and the offset accessed by each task. Currently, this information is available during task initialization step but is lost once initialization finishes and a dependency node is created. Adding this information helps us to easily identify the regions accessed just using information about the dependency node. Secondly, we augment each dependency hash table entry with a reference counter. The reference counter is incremented during dependency node initialization (__kmp_process_deps function) and is decremented when a task finishes execution and releases its dependencies (__kmp_release_deps function). When the reference count for a task reduces to zero the runtime system can infer that all the tasks that access a specific region have finished execution. Consequently the runtime utilizes this event as an appropriate point to signal to the hardware that the specific address region is not expected to be reused in the near future and can be marked as candidate for dead block eviction.

We have tested our implementation using several microbenchmarks each representing a different task DAG and with a representative kernel like SparseLU. Across the different test cases we have tested the implementation is able to correctly identify the last reuse of address regions in the application accurately. As a next step, we plan to extensively test our implementation with a broader set of microbenchmarks and kernels boasting complex task DAGs, integrate the changes to the runtime and generate traces for LLC accesses with and without dead region hints which can then be fed to the verification framework that we are developing as part of WP5. This would enable us to measure the potential benefit that DBM can provide in terms of reducing the LLC miss rate for real kernels.

6. Tools for debugging and monitoring

This section outlines work in WP4 related to tool support enabling debugging, performance monitoring and analysis, and power monitoring.

6.1 Extrae (trace capture)

We have successfully ported Extrae to the RISC-V architecture, allowing it to capture the parallel activity of the application, including references to the used runtimes. We also have developed an interface to read user accessible hardware counters, leaving it prepared to easily add the configurable counters requiring a higher access privilege. These improvements have been successfully tested on commercial SiFive boards (Unleashed and Unmatched), and initially deployed on an internally available development board. Changes in Extrae are already published in the main branch, available at https://github.com/bsc-performance-tools/extrae. The source code for the interface to read hardware counters, still under development, is also available at https://github.com/bsc-performance-tools/riscv-hwc-interface.

The following two figures illustrate timelines captured with Extrae and visualized with Paraver tool for an execution of the LULESH microbenchmark: (i) MPI activity (in Figure 7), and (ii) sampled functions (in Figure 8).

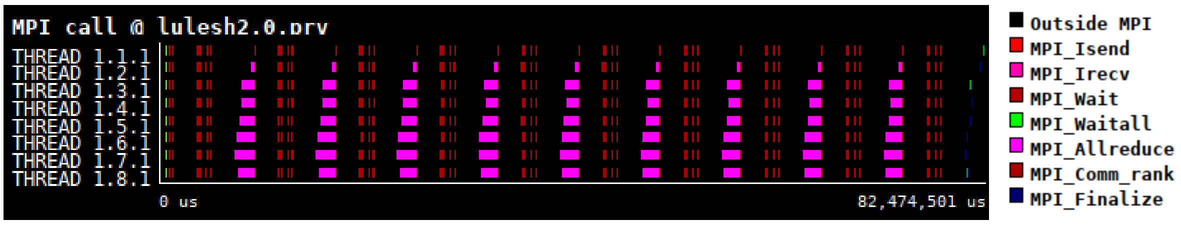


Figure 7: Paraver timeline showing the MPI activity of 10 iterations of the LULESH microbenchmark.

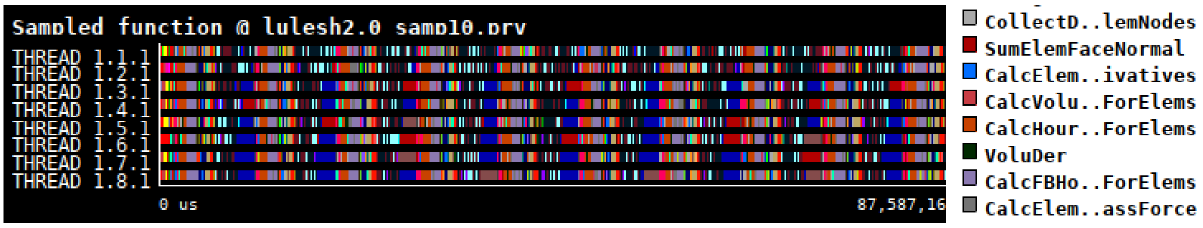


Figure 8: Paraver timeline showing the sampled functions using a frequency of 10ms.

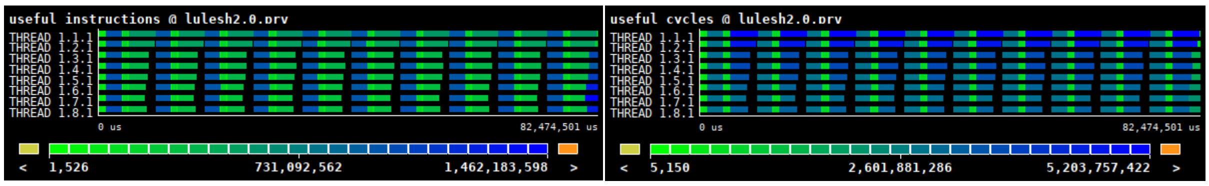


Figure 9: Number of instructions and number of cycles for each computation phase of LULESH.

Once access to privileged performance counters is available on eProcessor prototypes, we will extend Extrae and the interface to allow selecting which counters to read and write them into a Paraver tracefile to visually analyze and correlate them with the activity of the application.

6.2 GDB (debugger)

Following the work plan set in eProcessor Deliverable 3.1, we have successfully installed and tested gdb binary utility on the QEMU image of eProcessor. Native gdb testing is successful, while the testsuite of gdb shows high readiness to deploy. We are continuously working through failing tests in the gdb testsuite to ensure full functionality of the tool. A current snapshot of gdb, as deployed on QEMU, can be found at: https://b2drop.bsc.es/index.php/s/EN5XerJYjHeXeAH.

Next steps include (i) extending gdb with eProcessor vector instructions and CSRs, and (ii) using gdb with eProcessor applications, to increase the coverage of the testing process.

6.3 DynamoRIO (dynamic binary instrumentation)

DynamoRIO is a dynamic binary instrumentation tool, which allows monitoring and dynamically altering the behavior of a binary at execution time. It consists of three main elements:

- Core binary instrumentation mechanism;
- DynamoRIO API, which exposes the core functionality to application developers;
- Set of tools (debugging, simulation, tracing, code coverage, etc) built upon the core mechanism.

Currently the RISC-V porting of DynamoRIO is in progress. The build scripts and part of the instruction set encoding, have been ported, within the eProcessor project. Over the past months though, there has been a lot of activity in the DynamoRIO community of developers, towards porting the tool for the RISC-V architecture. This activity has materialized, with a few patches already committed. Given this opportunity, the eProcessor members working on porting DynamoRIO have already reached out to the corresponding developer community and have started collaborating with the community developers, in order to achieve the best outcome without duplication of effort.

6.4 Power monitoring

The Companion FPGA will also be used as the board controller for the eProcessor microserver module. A bare metal firmware will be running on one of the Cortex-R5 cores of the RPU (see Figure 4), managing, e.g., power sequencing of the eProcessor chip in response to external COM-HPC control signals. The firmware will also monitor voltage levels and the current consumption of a multitude of power rails on the eProcessor microserver module. Temperature sensors on the module as well as in the eProcessor itself will also be monitored. Table 2 shows the voltage rails of the eProcessor as well as those of the Companion FPGA that are currently planned to be monitored. Some of them will only be checked for a valid voltage level ("Power Good"), while the others will be connected to the internal ADC of the Companion FPGA to measure voltage and current. The FPGA's ADC has a maximum sample rate of 1 MSPS that will be shared between active measurement channels.

Table 2: Voltage rails of eProcessor and Companion FPGA to be monitored.

Rail	**Voltage**	**Measurement**
eProcessor		
CORE_1	0.9 V	Voltage, Current (individually for each sub-rail)
CORE_2		
ACCEL		
NOC, C2C		
DDR	1.2 V	Voltage, Current

DDR DLL	3.3 V	Power Good
C2C AVDD	0.9 V	Voltage, Current
C2C AVDDIO	1.8 V	
LVDS	1.8 V	Power Good
I/O		
PLL	3.3 V	Power Good
Module input	12 V	Voltage, Current
Companion FPGA		
CORE	0.85 V	Voltage, Current
CORE AUX	1.8 V	Power Good
DDR (2x)	1.2 V	Voltage, Current
C2C AVCC	0.9 V	Voltage, Current
C2C AVTT	1.2 V	
C2C AUX	1.8 V	
LVDS	1.8 V	Power Good
I/O	1.8 V, 3.3 V	Power Good

It is currently planned to have monitoring data available in two modes of operation: For regular monitoring tasks, which only require a relatively low update rate, the firmware will have a running average of the voltages and currents of the diverse power rails. To facilitate more fine-grained power measurements, e.g., for characterizing computational kernels, it is envisioned to implement an “oscilloscope mode” where a certain amount of data can be captured after a trigger event with a much higher sampling rate. Captured data can then afterwards be downloaded for analysis.

To integrate the regular monitoring with Linux, it is currently planned to implement a hardware monitoring driver, thus making the averages accessible in the /sys/class/hwmon/ hierarchy. For communication with the management firmware in the RPU an Xilinx LogiCORE IP Mailbox will be instantiated in the PL so that both processors can access it. The kernel sources already include a Linux driver for this Mailbox type.

The COM-HPC standard (to which the eProcessor microserver module will adhere to) also allows for out-of-band management via IPMI over dedicated IPMB pins or over Ethernet. It is foreseen to expose the high-speed measurements via this IPMI interface directly from the management firmware in the RPU. But also the regular monitoring data can be exposed as IPMI sensors. This way, the sensor data can be used by both: The OS running on the eProcessor (by using e.g., the “ipmitool” program) or an external management system on the carrier where the eProcessor microserver module will be installed.

7. Concluding remarks

This document provides a summary of work carried out in the context of WP4, as an accompaniment of the first release of the O/S, boot environment, compiler, and performance tools.

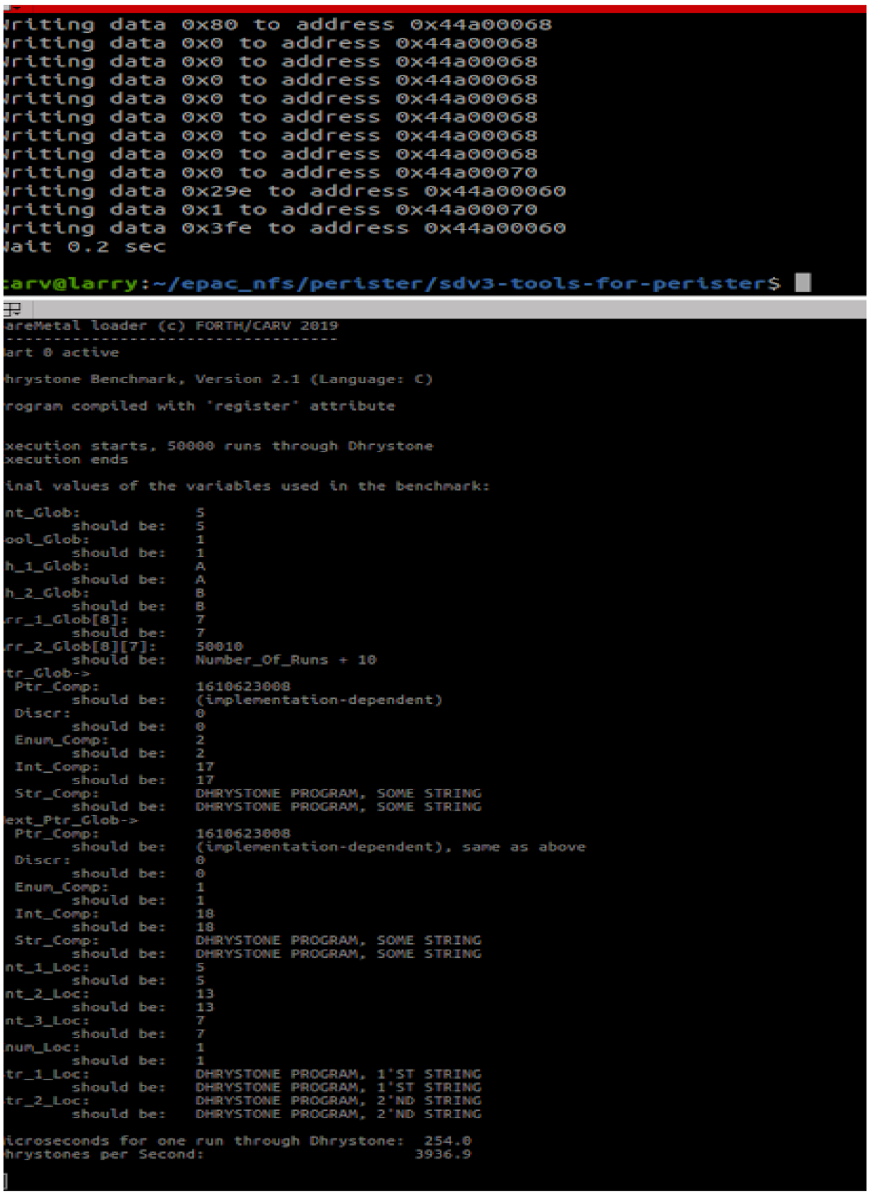
The effort documented in this deliverable provides a solid basis for advanced Linux support on eProcessor prototypes when the main core matures to the point of running Linux becomes feasible. The bulk of our work so far is expected to work as-is on the upcoming prototypes of eProcessor, with some adjustments in low-level firmware and OS code to handle specific details of the initialization sequence for the processing core. It will also help us address eProcessor specific aspects of the system, such as the boot cycle and the offloading to accelerators (since those are the functionalities that require special handling and otherwise a stock/vanilla kernel would suffice).

Follow-up work will be tracking the maturation of essential eProcessor hardware blocks (esp. of the main processing core), with the aim to finalize the support for a full Linux OS environment. This follow-up work is to be reported in D4.3 (“Release of the O/S, boot environment, compiler and performance tools for the single-core fabricated design” - due by M21).

8. Appendix

In this Appendix we provide (i) screenshots from the work on OS support, and (ii) evaluation results from a case study in the topic areas of software-based fault tolerance.

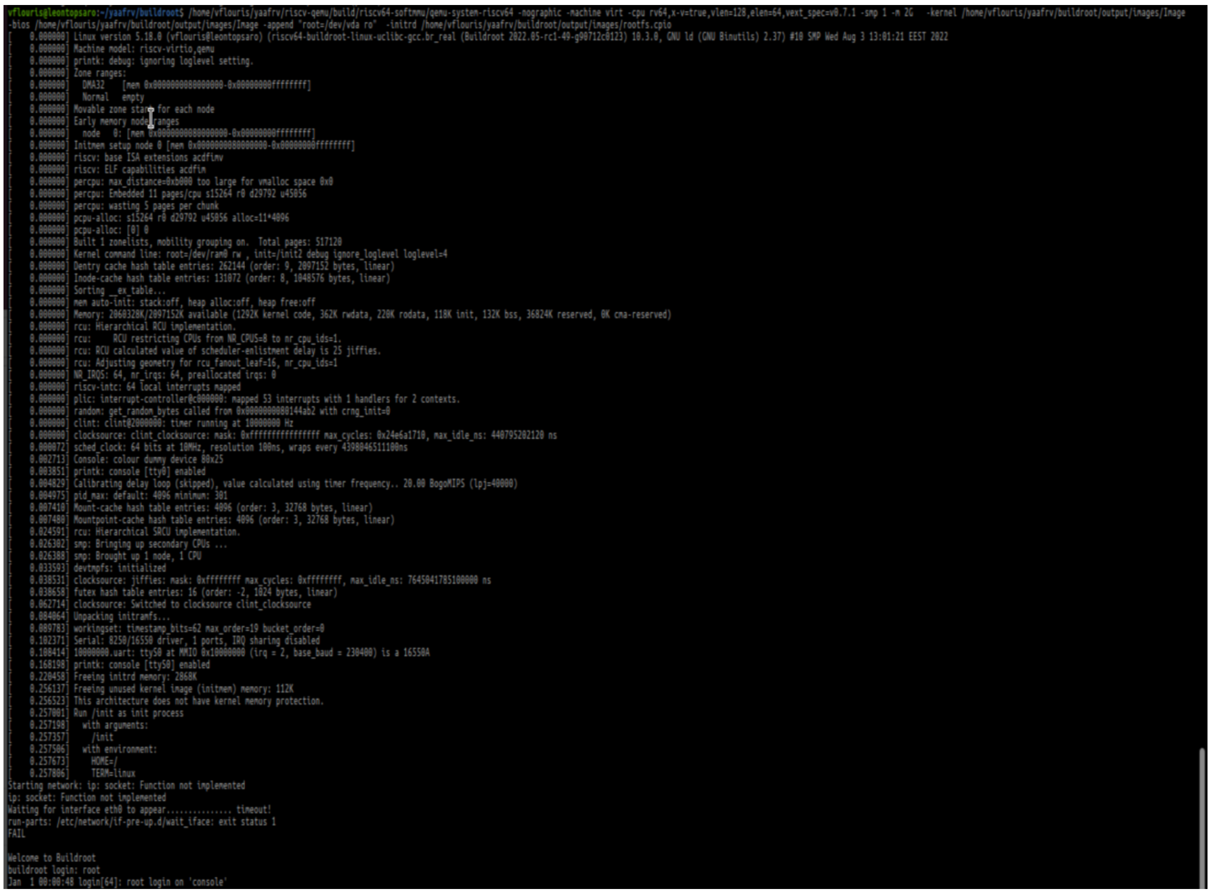
8.1 Screenshots from misc. operating modes on available platforms



```
Writing data 0x80 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00070
Writing data 0x29e to address 0x44a00060
Writing data 0x1 to address 0x44a00070
Writing data 0x3fe to address 0x44a00060
Wait 0.2 sec

carv@larry:~/epac_nfs/perister/sdv3-tools-for-perister$
```

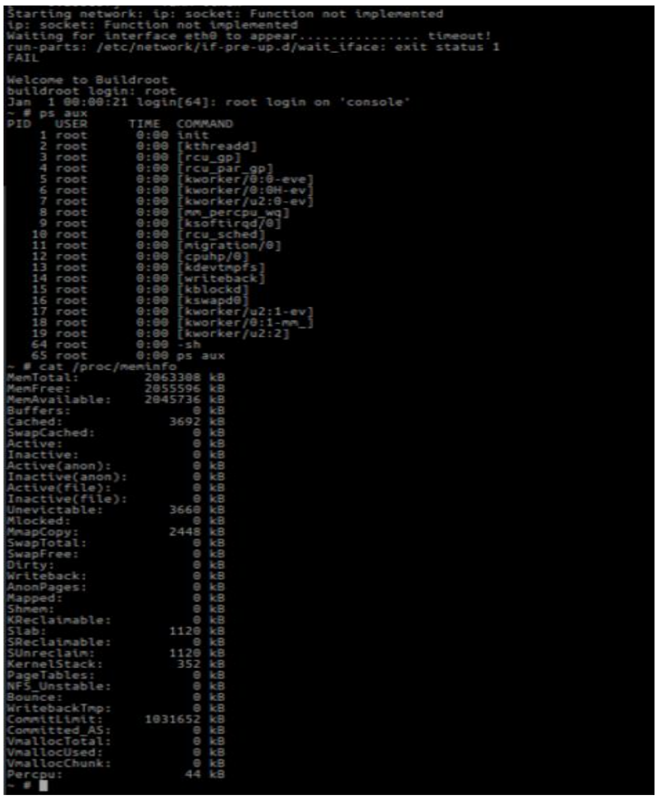
```
areMetal loader (c) FORTH/CARV 2019
----------------------------------------
art 0 active

hrystone Benchmark, Version 2.1 (Language: C)

rogram compiled with 'register' attribute


xecution starts, 50000 runs through Dhrystone
xecution ends

inal values of the variables used in the benchmark:

nt_Glob:            5
        should be:  5
ool_Glob:           1
        should be:  1
h_1_Glob:           A
        should be:  A
h_2_Glob:           B
        should be:  B
rr_1_Glob[8]:       7
        should be:  7
rr_2_Glob[8][7]:    50010
        should be:  Number_Of_Runs + 10
tr_Glob->
 Ptr_Comp:          1610623008
        should be:  (implementation-dependent)
 Discr:             0
        should be:  0
 Enum_Comp:         2
        should be:  2
 Int_Comp:          17
        should be:  17
 Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:  DHRYSTONE PROGRAM, SOME STRING
ext_Ptr_Glob->
 Ptr_Comp:          1610623008
        should be:  (implementation-dependent), same as above
 Discr:             0
        should be:  0
 Enum_Comp:         1
        should be:  1
 Int_Comp:          18
        should be:  18
 Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:  DHRYSTONE PROGRAM, SOME STRING
nt_1_Loc:           5
        should be:  5
nt_2_Loc:           13
        should be:  13
nt_3_Loc:           7
        should be:  7
num_Loc:            1
        should be:  1
tr_1_Loc:           DHRYSTONE PROGRAM, 1'ST STRING
        should be:  DHRYSTONE PROGRAM, 1'ST STRING
tr_2_Loc:           DHRYSTONE PROGRAM, 2'ND STRING
        should be:  DHRYSTONE PROGRAM, 2'ND STRING

icroseconds for one run through Dhrystone:  254.0
hrystones per Second:                      3936.9
```

Figure 10: Screenshot of Dhrystone benchmark run on EPI-SDV.



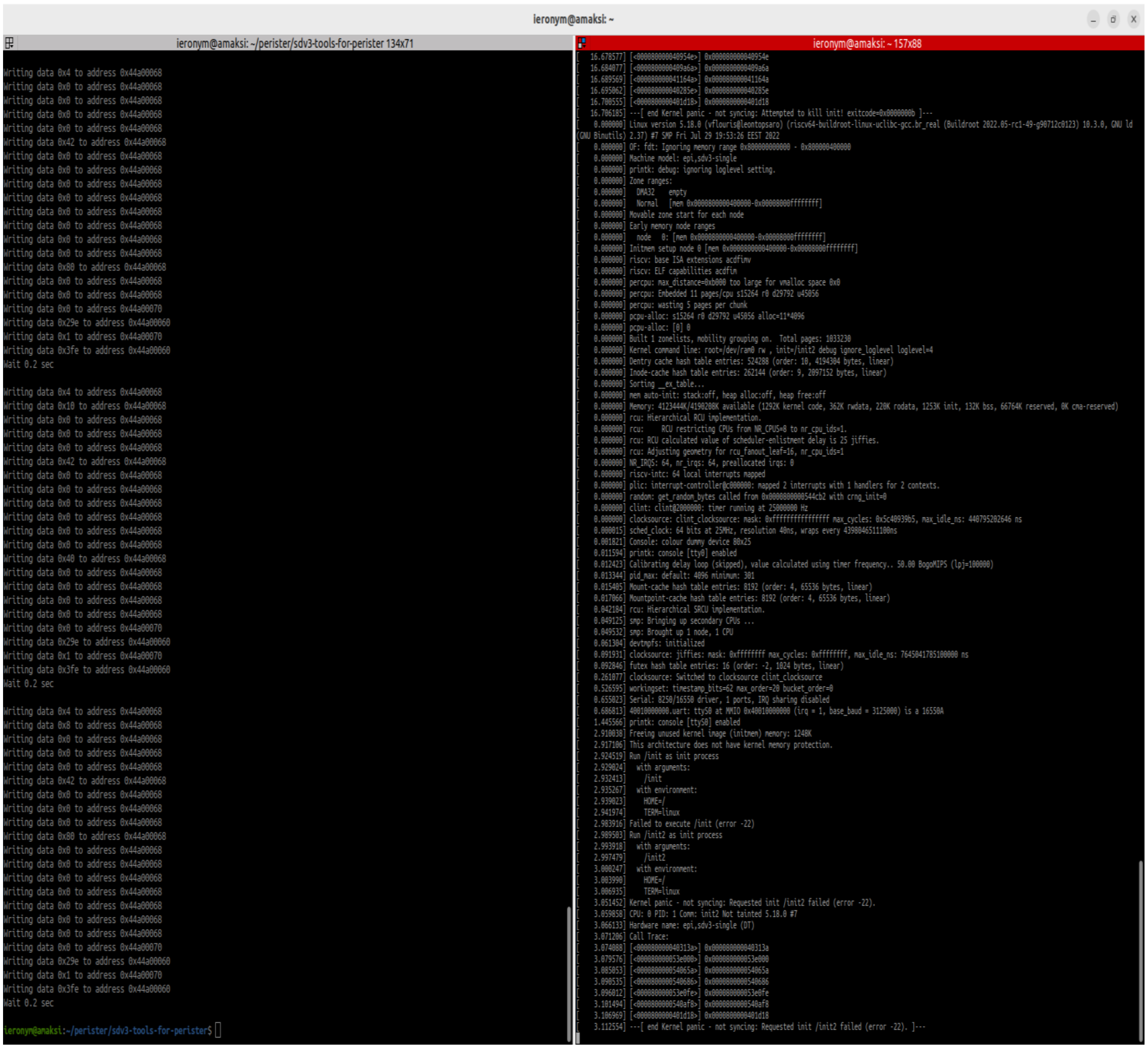
```
vflouris@leontopsaro:~/yaafrv/buildroot$ /home/vflouris/yaafrv/riscv-qemu/build/riscv64-softmmu/qemu-system-riscv64 -nographic -machine virt -cpu rv64,x-v=true,vlen=128,elen=64,vext_spec=v0.7.1 -smp 1 -m 2G  -kernel /home/vflouris/yaafrv/buildroot/output/images/Image
-bios /home/vflouris/yaafrv/buildroot/output/images/Image -append "root=/dev/vda ro"  -initrd /home/vflouris/yaafrv/buildroot/output/images/rootfs.cpio
[    0.000000] Linux version 5.18.0 (vflouris@leontopsaro) (riscv64-buildroot-linux-uclibc-gcc.br_real (Buildroot 2022.05-rc1-49-g90712c0123) 10.3.0, GNU ld (GNU Binutils) 2.37) #10 SMP Wed Aug 3 13:01:21 EEST 2022
[    0.000000] Machine model: riscv-virtio,qemu
[    0.000000] printk: debug: ignoring loglevel setting.
[    0.000000] Zone ranges:
[    0.000000]   DMA32    [mem 0x0000000080000000-0x00000000ffffffff]
[    0.000000]   Normal   empty
[    0.000000] Movable zone start for each node
[    0.000000] Early memory node ranges
[    0.000000]   node   0: [mem 0x0000000080000000-0x00000000ffffffff]
[    0.000000] Initmem setup node 0 [mem 0x0000000080000000-0x00000000ffffffff]
[    0.000000] riscv: base ISA extensions acdfimv
[    0.000000] riscv: ELF capabilities acdfim
[    0.000000] percpu: max_distance=0xb000 too large for vmalloc space 0x0
[    0.000000] percpu: Embedded 11 pages/cpu s15264 r0 d29792 u45056
[    0.000000] percpu: wasting 5 pages per chunk
[    0.000000] pcpu-alloc: s15264 r0 d29792 u45056 alloc=11*4096
[    0.000000] pcpu-alloc: [0] 0
[    0.000000] Built 1 zonelists, mobility grouping on.  Total pages: 517120
[    0.000000] Kernel command line: root=/dev/ram0 rw , init=/init2 debug ignore_loglevel loglevel=4
[    0.000000] Dentry cache hash table entries: 262144 (order: 9, 2097152 bytes, linear)
[    0.000000] Inode-cache hash table entries: 131072 (order: 8, 1048576 bytes, linear)
[    0.000000] Sorting __ex_table...
[    0.000000] mem auto-init: stack:off, heap alloc:off, heap free:off
[    0.000000] Memory: 2060328K/2097152K available (1292K kernel code, 362K rwdata, 220K rodata, 118K init, 132K bss, 36824K reserved, 0K cma-reserved)
[    0.000000] rcu: Hierarchical RCU implementation.
[    0.000000] rcu:     RCU restricting CPUs from NR_CPUS=8 to nr_cpu_ids=1.
[    0.000000] rcu: RCU calculated value of scheduler-enlistment delay is 25 jiffies.
[    0.000000] rcu: Adjusting geometry for rcu_fanout_leaf=16, nr_cpu_ids=1
[    0.000000] NR_IRQS: 64, nr_irqs: 64, preallocated irqs: 0
[    0.000000] riscv-intc: 64 local interrupts mapped
[    0.000000] plic: interrupt-controller@c000000: mapped 53 interrupts with 1 handlers for 2 contexts.
[    0.000000] random: get_random_bytes called from 0x0000000080144ab2 with crng_init=0
[    0.000000] clint: clint@2000000: timer running at 10000000 Hz
[    0.000000] clocksource: clint_clocksource: mask: 0xffffffffffffffff max_cycles: 0x24e6a1710, max_idle_ns: 440795202120 ns
[    0.000072] sched_clock: 64 bits at 10MHz, resolution 100ns, wraps every 4398046511100ns
[    0.002713] Console: colour dummy device 80x25
[    0.003851] printk: console [tty0] enabled
[    0.004829] Calibrating delay loop (skipped), value calculated using timer frequency.. 20.00 BogoMIPS (lpj=40000)
[    0.004975] pid_max: default: 4096 minimum: 301
[    0.007410] Mount-cache hash table entries: 4096 (order: 3, 32768 bytes, linear)
[    0.007480] Mountpoint-cache hash table entries: 4096 (order: 3, 32768 bytes, linear)
[    0.024591] rcu: Hierarchical SRCU implementation.
[    0.026302] smp: Bringing up secondary CPUs ...
[    0.026388] smp: Brought up 1 node, 1 CPU
[    0.033593] devtmpfs: initialized
[    0.038531] clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 7645041785100000 ns
[    0.038658] futex hash table entries: 16 (order: -2, 1024 bytes, linear)
[    0.062714] clocksource: Switched to clocksource clint_clocksource
[    0.084064] Unpacking initramfs...
[    0.089783] workingset: timestamp_bits=62 max_order=19 bucket_order=0
[    0.102371] Serial: 8250/16550 driver, 1 ports, IRQ sharing disabled
[    0.108414] 10000000.uart: ttyS0 at MMIO 0x10000000 (irq = 2, base_baud = 230400) is a 16550A
[    0.168198] printk: console [ttyS0] enabled
[    0.220458] Freeing initrd memory: 2868K
[    0.256137] Freeing unused kernel image (initmem) memory: 112K
[    0.256523] This architecture does not have kernel memory protection.
[    0.257001] Run /init as init process
[    0.257198]   with arguments:
[    0.257357]     /init
[    0.257506]   with environment:
[    0.257673]     HOME=/
[    0.257806]     TERM=linux
Starting network: ip: socket: Function not implemented
ip: socket: Function not implemented
Waiting for interface eth0 to appear............... timeout!
run-parts: /etc/network/if-pre-up.d/wait_iface: exit status 1
FAIL

Welcome to Buildroot
buildroot login: root
Jan  1 00:00:48 login[64]: root login on 'console'
```

Figure 11: Screenshot of nommu linux booting on qemu-riscv64.



```
Starting network: ip: socket: Function not implemented
ip: socket: Function not implemented
Waiting for interface eth0 to appear................ timeout!
run-parts: /etc/network/if-pre-up.d/wait_iface: exit status 1
FAIL

Welcome to Buildroot
buildroot login: root
Jan  1 00:00:21 login[64]: root login on 'console'
~ # ps aux
PID   USER     TIME  COMMAND
    1 root      0:00 init
    2 root      0:00 [kthreadd]
    3 root      0:00 [rcu_gp]
    4 root      0:00 [rcu_par_gp]
    5 root      0:00 [kworker/0:0-eve]
    6 root      0:00 [kworker/0:0H-ev]
    7 root      0:00 [kworker/u2:0-ev]
    8 root      0:00 [mm_percpu_wq]
    9 root      0:00 [ksoftirqd/0]
   10 root      0:00 [rcu_sched]
   11 root      0:00 [migration/0]
   12 root      0:00 [cpuhp/0]
   13 root      0:00 [kdevtmpfs]
   14 root      0:00 [writeback]
   15 root      0:00 [kblockd]
   16 root      0:00 [kswapd0]
   17 root      0:00 [kworker/u2:1-ev]
   18 root      0:00 [kworker/0:1-mm_]
   19 root      0:00 [kworker/u2:2]
   64 root      0:00 -sh
   65 root      0:00 ps aux
~ # cat /proc/meminfo
MemTotal:        2063308 kB
MemFree:         2055596 kB
MemAvailable:    2045736 kB
Buffers:               0 kB
Cached:             3692 kB
SwapCached:            0 kB
Active:                0 kB
Inactive:              0 kB
Active(anon):          0 kB
Inactive(anon):        0 kB
Active(file):          0 kB
Inactive(file):        0 kB
Unevictable:        3660 kB
Mlocked:               0 kB
MmapCopy:           2448 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:                 0 kB
Writeback:             0 kB
AnonPages:             0 kB
Mapped:                0 kB
Shmem:                 0 kB
KReclaimable:          0 kB
Slab:               1120 kB
SReclaimable:          0 kB
SUnreclaim:         1120 kB
KernelStack:         352 kB
PageTables:            0 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:     1031652 kB
Committed_AS:          0 kB
VmallocTotal:          0 kB
VmallocUsed:           0 kB
VmallocChunk:          0 kB
Percpu:               44 kB
~ # █
```

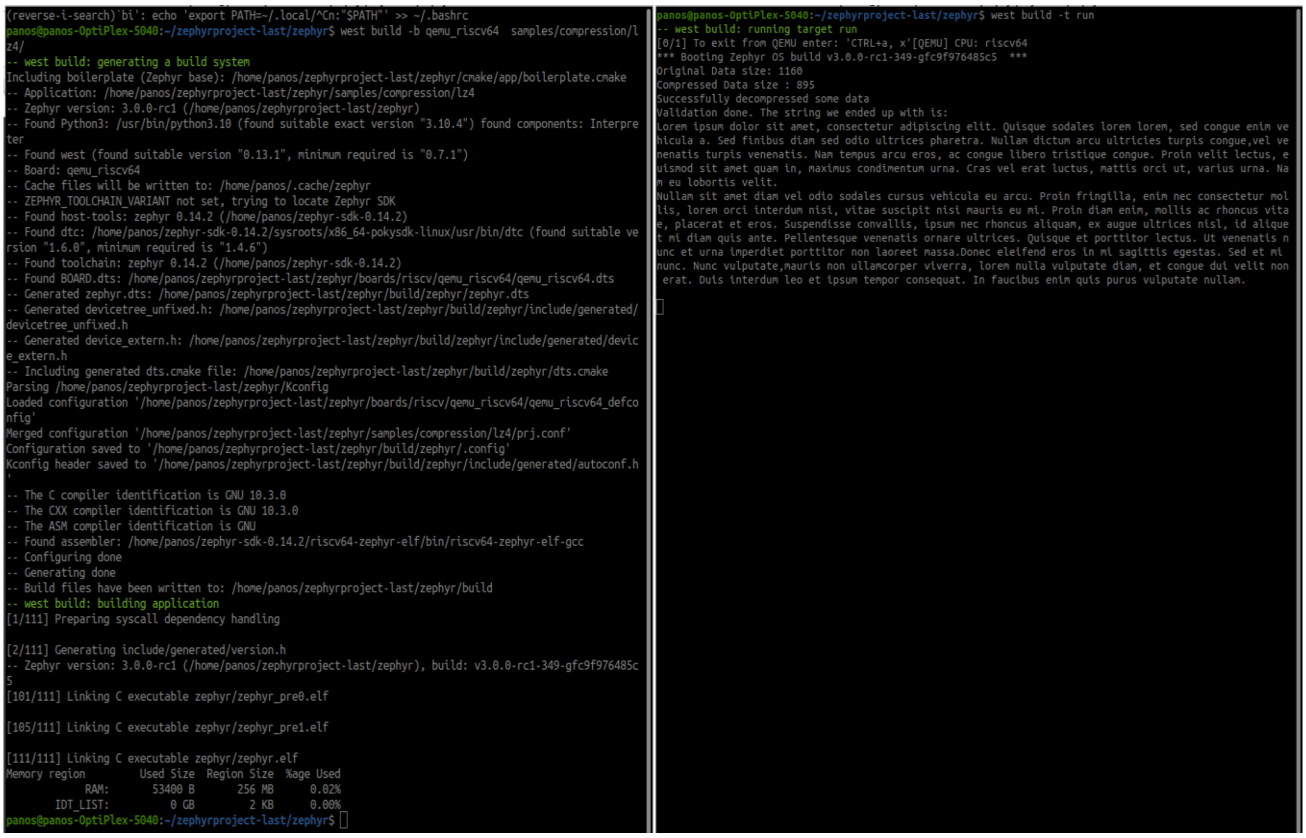
Figure 12: Screenshot of meminfo output in nommu linux running on qemu-riscv64.



```
Writing data 0x4 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x42 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x80 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00070
Writing data 0x29e to address 0x44a00060
Writing data 0x1 to address 0x44a00070
Writing data 0x3fe to address 0x44a00060
Wait 0.2 sec

Writing data 0x4 to address 0x44a00068
Writing data 0x10 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x42 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x40 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00070
Writing data 0x29e to address 0x44a00060
Writing data 0x1 to address 0x44a00070
Writing data 0x3fe to address 0x44a00060
Wait 0.2 sec

Writing data 0x4 to address 0x44a00068
Writing data 0x8 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x42 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x80 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00070
Writing data 0x29e to address 0x44a00060
Writing data 0x1 to address 0x44a00070
Writing data 0x3fe to address 0x44a00060
Wait 0.2 sec

ieronym@amaksi:~/perister/sdv3-tools-for-perister$
```

```
[   16.678577] [<000080000040954e>] 0x000080000040954e
[   16.684077] [<00008000004009a6a>] 0x0000800000409a6a
[   16.689569] [<000080000041164a>] 0x000080000041164a
[   16.695062] [<000080000040285e>] 0x000080000040285e
[   16.700555] [<0000800000401d18>] 0x0000800000401d18
[   16.706185] ---[ end Kernel panic - not syncing: Attempted to kill init! exitcode=0x0000000b ]---
[    0.000000] Linux version 5.18.0 (vflouris@leontopsaro) (riscv64-buildroot-linux-uclibc-gcc.br_real (Buildroot 2022.05-rc1-49-g90712c0123) 10.3.0, GNU ld
(GNU Binutils) 2.37) #7 SMP Fri Jul 29 19:53:26 EEST 2022
[    0.000000] OF: fdt: Ignoring memory range 0x800000000000 - 0x800000400000
[    0.000000] Machine model: epi,sdv3-single
[    0.000000] printk: debug: ignoring loglevel setting.
[    0.000000] Zone ranges:
[    0.000000]   DMA32    empty
[    0.000000]   Normal   [mem 0x0000800000400000-0x00008000ffffffff]
[    0.000000] Movable zone start for each node
[    0.000000] Early memory node ranges
[    0.000000]   node   0: [mem 0x0000800000400000-0x00008000ffffffff]
[    0.000000] Initmem setup node 0 [mem 0x0000800000400000-0x00008000ffffffff]
[    0.000000] riscv: base ISA extensions acdfimv
[    0.000000] riscv: ELF capabilities acdfim
[    0.000000] percpu: max_distance=0xb000 too large for vmalloc space 0x0
[    0.000000] percpu: Embedded 11 pages/cpu s15264 r0 d29792 u45056
[    0.000000] percpu: wasting 5 pages per chunk
[    0.000000] pcpu-alloc: s15264 r0 d29792 u45056 alloc=11*4096
[    0.000000] pcpu-alloc: [0] 0
[    0.000000] Built 1 zonelists, mobility grouping on.  Total pages: 1033230
[    0.000000] Kernel command line: root=/dev/ram0 rw , init=/init2 debug ignore_loglevel loglevel=4
[    0.000000] Dentry cache hash table entries: 524288 (order: 10, 4194304 bytes, linear)
[    0.000000] Inode-cache hash table entries: 262144 (order: 9, 2097152 bytes, linear)
[    0.000000] Sorting __ex_table...
[    0.000000] mem auto-init: stack:off, heap alloc:off, heap free:off
[    0.000000] Memory: 4123444K/4190208K available (1292K kernel code, 362K rwdata, 220K rodata, 1253K init, 132K bss, 66764K reserved, 0K cma-reserved)
[    0.000000] rcu: Hierarchical RCU implementation.
[    0.000000] rcu:     RCU restricting CPUs from NR_CPUS=8 to nr_cpu_ids=1.
[    0.000000] rcu: RCU calculated value of scheduler-enlistment delay is 25 jiffies.
[    0.000000] rcu: Adjusting geometry for rcu_fanout_leaf=16, nr_cpu_ids=1
[    0.000000] NR_IRQS: 64, nr_irqs: 64, preallocated irqs: 0
[    0.000000] riscv-intc: 64 local interrupts mapped
[    0.000000] plic: interrupt-controller@c000000: mapped 2 interrupts with 1 handlers for 2 contexts.
[    0.000000] random: get_random_bytes called from 0x0000800000544cb2 with crng_init=0
[    0.000000] clint: clint@2000000: timer running at 25000000 Hz
[    0.000000] clocksource: clint_clocksource: mask: 0xffffffffffffffff max_cycles: 0x5c40939b5, max_idle_ns: 440795202646 ns
[    0.000015] sched_clock: 64 bits at 25MHz, resolution 40ns, wraps every 4398046511100ns
[    0.001821] Console: colour dummy device 80x25
[    0.011594] printk: console [tty0] enabled
[    0.012423] Calibrating delay loop (skipped), value calculated using timer frequency.. 50.00 BogoMIPS (lpj=100000)
[    0.013344] pid_max: default: 4096 minimum: 301
[    0.015405] Mount-cache hash table entries: 8192 (order: 4, 65536 bytes, linear)
[    0.017066] Mountpoint-cache hash table entries: 8192 (order: 4, 65536 bytes, linear)
[    0.042184] rcu: Hierarchical SRCU implementation.
[    0.049125] smp: Bringing up secondary CPUs ...
[    0.049532] smp: Brought up 1 node, 1 CPU
[    0.061304] devtmpfs: initialized
[    0.091931] clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 7645041785100000 ns
[    0.092846] futex hash table entries: 16 (order: -2, 1024 bytes, linear)
[    0.261077] clocksource: Switched to clocksource clint_clocksource
[    0.526595] workingset: timestamp_bits=62 max_order=20 bucket_order=0
[    0.655023] Serial: 8250/16550 driver, 1 ports, IRQ sharing disabled
[    0.686813] 40010000000.uart: ttyS0 at MMIO 0x40010000000 (irq = 1, base_baud = 3125000) is a 16550A
[    1.445566] printk: console [ttyS0] enabled
[    2.910038] Freeing unused kernel image (initmem) memory: 1248K
[    2.917106] This architecture does not have kernel memory protection.
[    2.924519] Run /init as init process
[    2.929024]   with arguments:
[    2.932413]     /init
[    2.935267]   with environment:
[    2.939023]     HOME=/
[    2.941974]     TERM=linux
[    2.983916] Failed to execute /init (error -22)
[    2.989503] Run /init2 as init process
[    2.993918]   with arguments:
[    2.997479]     /init2
[    3.000247]   with environment:
[    3.003990]     HOME=/
[    3.006935]     TERM=linux
[    3.051452] Kernel panic - not syncing: Requested init /init2 failed (error -22).
[    3.059858] CPU: 0 PID: 1 Comm: init2 Not tainted 5.18.0 #7
[    3.066133] Hardware name: epi,sdv3-single (DT)
[    3.071206] Call Trace:
[    3.074088] [<000080000040313a>] 0x000080000040313a
[    3.079576] [<000080000053e000>] 0x000080000053e000
[    3.085053] [<000080000054065a>] 0x000080000054065a
[    3.090535] [<0000800000540686>] 0x0000800000540686
[    3.096012] [<000080000053e0fe>] 0x000080000053e0fe
[    3.101494] [<0000800000540af8>] 0x0000800000540af8
[    3.106969] [<0000800000401d18>] 0x0000800000401d18
[    3.112554] ---[ end Kernel panic - not syncing: Requested init /init2 failed (error -22). ]---
```

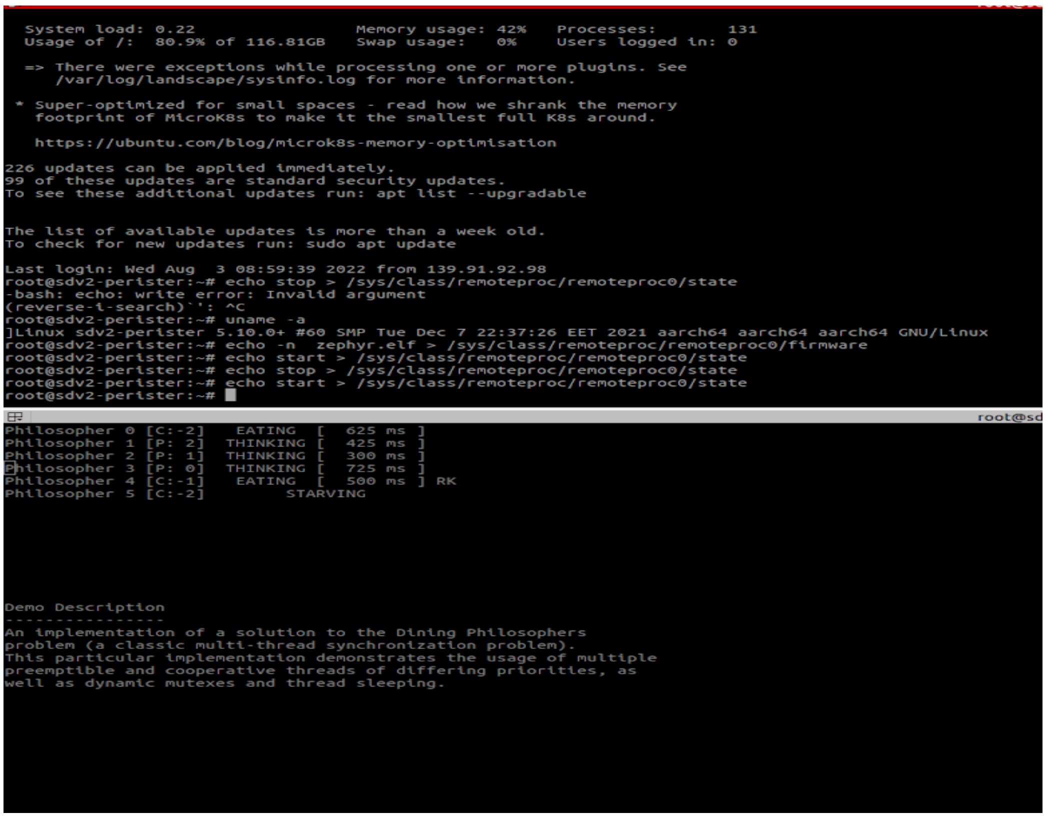
Figure 13: Screenshot of nommu-linux running BusyBox on EPI-SDV.



```
(reverse-i-search)`bi': echo 'export PATH=~/.local/^Cn:"$PATH"' >> ~/.bashrc
panos@panos-OptiPlex-5040:~/zephyrproject-last/zephyr$ west build -b qemu_riscv64  samples/compression/l
z4/
-- west build: generating a build system
Including boilerplate (Zephyr base): /home/panos/zephyrproject-last/zephyr/cmake/app/boilerplate.cmake
-- Application: /home/panos/zephyrproject-last/zephyr/samples/compression/lz4
-- Zephyr version: 3.0.0-rc1 (/home/panos/zephyrproject-last/zephyr)
-- Found Python3: /usr/bin/python3.10 (found suitable exact version "3.10.4") found components: Interpre
ter
-- Found west (found suitable version "0.13.1", minimum required is "0.7.1")
-- Board: qemu_riscv64
-- Cache files will be written to: /home/panos/.cache/zephyr
-- ZEPHYR_TOOLCHAIN_VARIANT not set, trying to locate Zephyr SDK
-- Found host-tools: zephyr 0.14.2 (/home/panos/zephyr-sdk-0.14.2)
-- Found dtc: /home/panos/zephyr-sdk-0.14.2/sysroots/x86_64-pokysdk-linux/usr/bin/dtc (found suitable ve
rsion "1.6.0", minimum required is "1.4.6")
-- Found toolchain: zephyr 0.14.2 (/home/panos/zephyr-sdk-0.14.2)
-- Found BOARD.dts: /home/panos/zephyrproject-last/zephyr/boards/riscv/qemu_riscv64/qemu_riscv64.dts
-- Generated zephyr.dts: /home/panos/zephyrproject-last/zephyr/build/zephyr/zephyr.dts
-- Generated devicetree_unfixed.h: /home/panos/zephyrproject-last/zephyr/build/zephyr/include/generated/
devicetree_unfixed.h
-- Generated device_extern.h: /home/panos/zephyrproject-last/zephyr/build/zephyr/include/generated/devic
e_extern.h
-- Including generated dts.cmake file: /home/panos/zephyrproject-last/zephyr/build/zephyr/dts.cmake
Parsing /home/panos/zephyrproject-last/zephyr/Kconfig
Loaded configuration '/home/panos/zephyrproject-last/zephyr/boards/riscv/qemu_riscv64/qemu_riscv64_defco
nfig'
Merged configuration '/home/panos/zephyrproject-last/zephyr/samples/compression/lz4/prj.conf'
Configuration saved to '/home/panos/zephyrproject-last/zephyr/build/zephyr/.config'
Kconfig header saved to '/home/panos/zephyrproject-last/zephyr/build/zephyr/include/generated/autoconf.h
'

-- The C compiler identification is GNU 10.3.0
-- The CXX compiler identification is GNU 10.3.0
-- The ASM compiler identification is GNU
-- Found assembler: /home/panos/zephyr-sdk-0.14.2/riscv64-zephyr-elf/bin/riscv64-zephyr-elf-gcc
-- Configuring done
-- Generating done
-- Build files have been written to: /home/panos/zephyrproject-last/zephyr/build
-- west build: building application
[1/111] Preparing syscall dependency handling

[2/111] Generating include/generated/version.h
-- Zephyr version: 3.0.0-rc1 (/home/panos/zephyrproject-last/zephyr), build: v3.0.0-rc1-349-gfc9f976485c
5
[101/111] Linking C executable zephyr/zephyr_pre0.elf

[105/111] Linking C executable zephyr/zephyr_pre1.elf

[111/111] Linking C executable zephyr/zephyr.elf
Memory region         Used Size  Region Size  %age Used
             RAM:       53400 B       256 MB      0.02%
        IDT_LIST:          0 GB         2 KB      0.00%
panos@panos-OptiPlex-5040:~/zephyrproject-last/zephyr$
```

```
panos@panos-OptiPlex-5040:~/zephyrproject-last/zephyr$ west build -t run
-- west build: running target run
[0/1] To exit from QEMU enter: 'CTRL+a, x'[QEMU] CPU: riscv64
*** Booting Zephyr OS build v3.0.0-rc1-349-gfc9f976485c5  ***
Original Data size: 1160
Compressed Data size : 895
Successfully decompressed some data
Validation done. The string we ended up with is:
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Quisque sodales lorem lorem, sed congue enim ve
hicula a. Sed finibus diam sed odio ultrices pharetra. Nullam dictum arcu ultricies turpis congue,vel ve
nenatis turpis venenatis. Nam tempus arcu eros, ac congue libero tristique congue. Proin velit lectus, e
uismod sit amet quam in, maximus condimentum urna. Cras vel erat luctus, mattis orci ut, varius urna. Na
m eu lobortis velit.
Nullam sit amet diam vel odio sodales cursus vehicula eu arcu. Proin fringilla, enim nec consectetur mol
lis, lorem orci interdum nisi, vitae suscipit nisi mauris eu mi. Proin diam enim, mollis ac rhoncus vita
e, placerat et eros. Suspendisse convallis, ipsum nec rhoncus aliquam, ex augue ultrices nisl, id alique
t mi diam quis ante. Pellentesque venenatis ornare ultrices. Quisque et porttitor lectus. Ut venenatis n
unc et urna imperdiet porttitor non laoreet massa.Donec eleifend eros in mi sagittis egestas. Sed et mi
nunc. Nunc vulputate,mauris non ullamcorper viverra, lorem nulla vulputate diam, et congue dui velit non
 erat. Duis interdum leo et ipsum tempor consequat. In faucibus enim quis purus vulputate nullam.
```

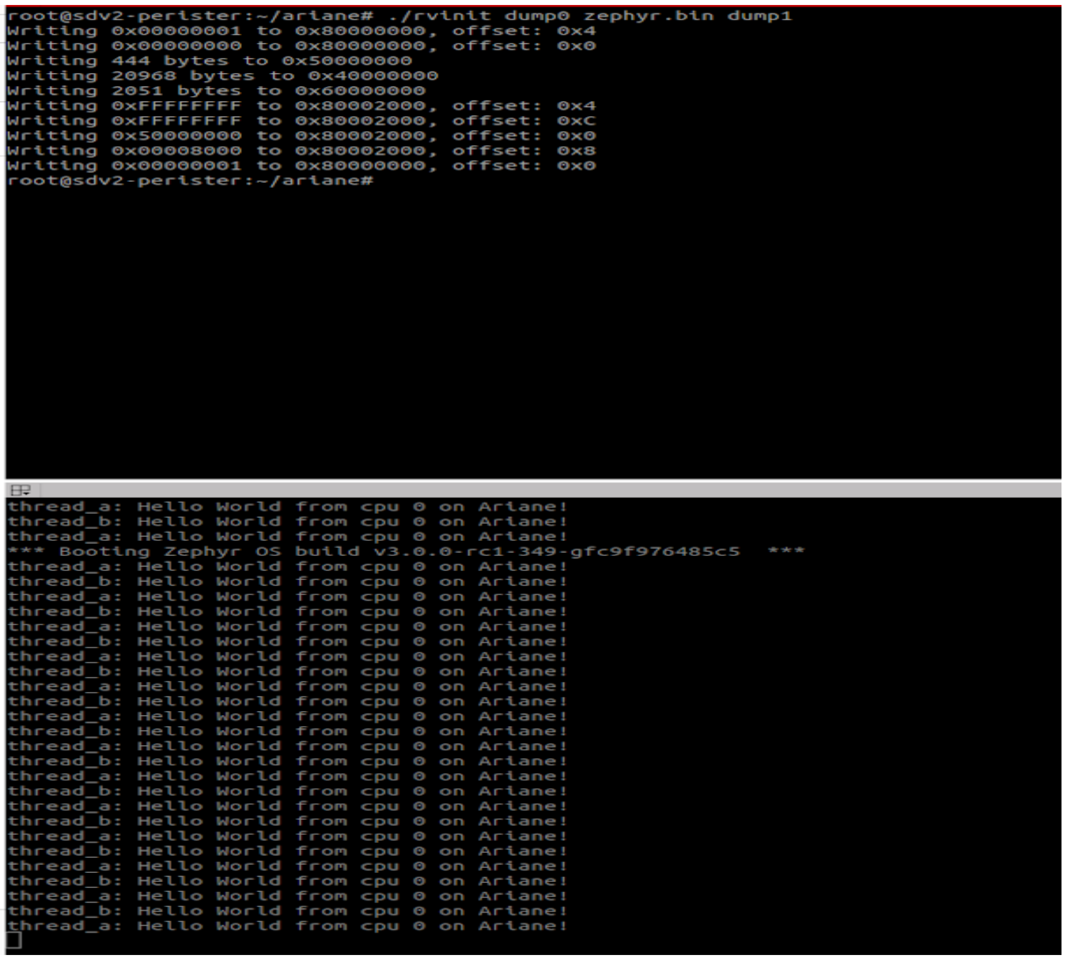
Figure 14: Screenshot of a Zephyr compression application, being build and then executing on qemu riscv64 emulator.



```
  System load:  0.22                Memory usage: 42%   Processes:        131
  Usage of /:   80.9% of 116.81GB   Swap usage:   0%    Users logged in: 0

  => There were exceptions while processing one or more plugins. See
     /var/log/landscape/sysinfo.log for more information.

 * Super-optimized for small spaces - read how we shrank the memory
   footprint of MicroK8s to make it the smallest full K8s around.

   https://ubuntu.com/blog/microk8s-memory-optimisation

226 updates can be applied immediately.
99 of these updates are standard security updates.
To see these additional updates run: apt list --upgradable


The list of available updates is more than a week old.
To check for new updates run: sudo apt update

Last login: Wed Aug  3 08:59:39 2022 from 139.91.92.98
root@sdv2-perister:~# echo stop > /sys/class/remoteproc/remoteproc0/state
-bash: echo: write error: Invalid argument
(reverse-i-search)`': ^C
root@sdv2-perister:~# uname -a
]Linux sdv2-perister 5.10.0+ #60 SMP Tue Dec 7 22:37:26 EET 2021 aarch64 aarch64 aarch64 GNU/Linux
root@sdv2-perister:~# echo -n  zephyr.elf > /sys/class/remoteproc/remoteproc0/firmware
root@sdv2-perister:~# echo start > /sys/class/remoteproc/remoteproc0/state
root@sdv2-perister:~# echo stop > /sys/class/remoteproc/remoteproc0/state
root@sdv2-perister:~# echo start > /sys/class/remoteproc/remoteproc0/state
root@sdv2-perister:~#
```

```
Philosopher 0 [C:-2]   EATING  [  625 ms ]
Philosopher 1 [P: 2]  THINKING [  425 ms ]
Philosopher 2 [P: 1]  THINKING [  300 ms ]
Philosopher 3 [P: 0]  THINKING [  725 ms ]
Philosopher 4 [C:-1]   EATING  [  500 ms ] RK
Philosopher 5 [C:-2]         STARVING




Demo Description
----------------
An implementation of a solution to the Dining Philosophers
problem (a classic multi-thread synchronization problem).
This particular implementation demonstrates the usage of multiple
preemptible and cooperative threads of differing priorities, as
well as dynamic mutexes and thread sleeping.
```

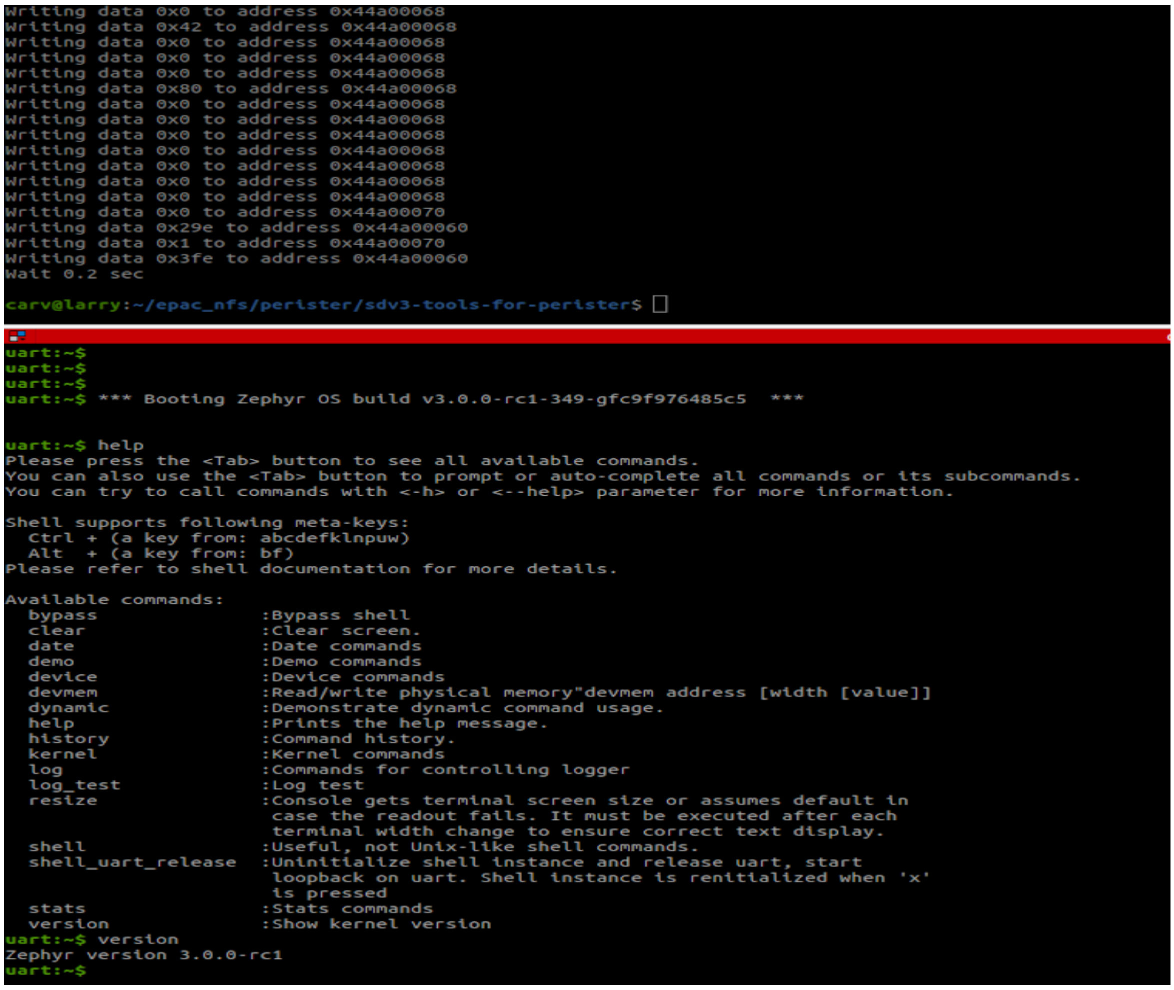
Figure 15: Screenshot of philosophers dining problem Zephyr application running on Ariane-SDV using remoteproc.



```
root@sdv2-perister:~/ariane# ./rvinit dump0 zephyr.bin dump1
Writing 0x00000001 to 0x80000000, offset: 0x4
Writing 0x00000000 to 0x80000000, offset: 0x0
Writing 444 bytes to 0x50000000
Writing 20968 bytes to 0x40000000
Writing 2051 bytes to 0x60000000
Writing 0xFFFFFFFF to 0x80002000, offset: 0x4
Writing 0xFFFFFFFF to 0x80002000, offset: 0xC
Writing 0x50000000 to 0x80002000, offset: 0x0
Writing 0x00008000 to 0x80002000, offset: 0x8
Writing 0x00000001 to 0x80000000, offset: 0x0
root@sdv2-perister:~/ariane#
```

```
thread_a: Hello World from cpu 0 on Ariane!
thread_b: Hello World from cpu 0 on Ariane!
thread_a: Hello World from cpu 0 on Ariane!
*** Booting Zephyr OS build v3.0.0-rc1-349-gfc9f976485c5  ***
thread_a: Hello World from cpu 0 on Ariane!
thread_b: Hello World from cpu 0 on Ariane!
thread_a: Hello World from cpu 0 on Ariane!
thread_b: Hello World from cpu 0 on Ariane!
thread_a: Hello World from cpu 0 on Ariane!
thread_b: Hello World from cpu 0 on Ariane!
thread_a: Hello World from cpu 0 on Ariane!
thread_b: Hello World from cpu 0 on Ariane!
thread_a: Hello World from cpu 0 on Ariane!
thread_b: Hello World from cpu 0 on Ariane!
thread_a: Hello World from cpu 0 on Ariane!
thread_b: Hello World from cpu 0 on Ariane!
thread_a: Hello World from cpu 0 on Ariane!
thread_b: Hello World from cpu 0 on Ariane!
thread_a: Hello World from cpu 0 on Ariane!
thread_b: Hello World from cpu 0 on Ariane!
thread_a: Hello World from cpu 0 on Ariane!
thread_b: Hello World from cpu 0 on Ariane!
thread_a: Hello World from cpu 0 on Ariane!
thread_b: Hello World from cpu 0 on Ariane!
thread_a: Hello World from cpu 0 on Ariane!
thread_b: Hello World from cpu 0 on Ariane!
thread_a: Hello World from cpu 0 on Ariane!
thread_b: Hello World from cpu 0 on Ariane!
thread_a: Hello World from cpu 0 on Ariane!
```

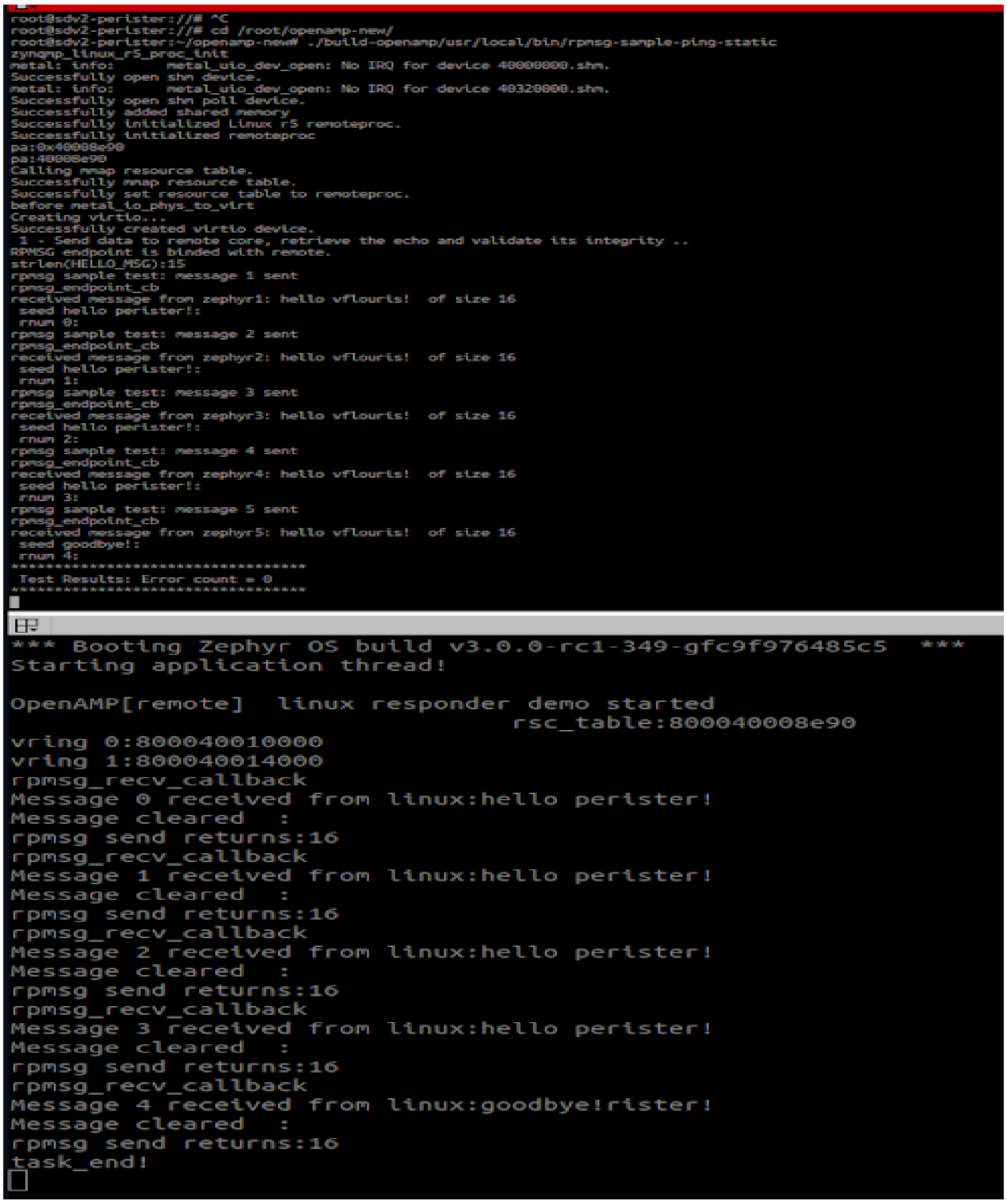
Figure 16: Screenshot of thread synchronization Zephyr application running on Ariane-SDV, via rvinit.



```
Writing data 0x0 to address 0x44a00068
Writing data 0x42 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x80 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00068
Writing data 0x0 to address 0x44a00070
Writing data 0x29e to address 0x44a00060
Writing data 0x1 to address 0x44a00070
Writing data 0x3fe to address 0x44a00060
Wait 0.2 sec

carv@larry:~/epac_nfs/perister/sdv3-tools-for-perister$
```

```
uart:~$
uart:~$
uart:~$
uart:~$ *** Booting Zephyr OS build v3.0.0-rc1-349-gfc9f976485c5  ***


uart:~$ help
Please press the <Tab> button to see all available commands.
You can also use the <Tab> button to prompt or auto-complete all commands or its subcommands.
You can try to call commands with <-h> or <--help> parameter for more information.

Shell supports following meta-keys:
  Ctrl + (a key from: abcdefklnpuw)
  Alt  + (a key from: bf)
Please refer to shell documentation for more details.

Available commands:
  bypass              :Bypass shell
  clear               :Clear screen.
  date                :Date commands
  demo                :Demo commands
  device              :Device commands
  devmem              :Read/write physical memory"devmem address [width [value]]
  dynamic             :Demonstrate dynamic command usage.
  help                :Prints the help message.
  history             :Command history.
  kernel              :Kernel commands
  log                 :Commands for controlling logger
  log_test            :Log test
  resize              :Console gets terminal screen size or assumes default in
                       case the readout fails. It must be executed after each
                       terminal width change to ensure correct text display.
  shell               :Useful, not Unix-like shell commands.
  shell_uart_release  :Uninitialize shell instance and release uart, start
                       loopback on uart. Shell instance is renitialized when 'x'
                       is pressed
  stats               :Stats commands
  version             :Show kernel version
uart:~$ version
Zephyr version 3.0.0-rc1
uart:~$
```

Figure 17: Screenshot of Zephyr shell application on EPI-SDV, which runs build-in (e.g devmem) as well as user-defined commands.



```
root@sdv2-perister://# ^C
root@sdv2-perister://# cd /root/openamp-new/
root@sdv2-perister:~/openamp-new# ./build-openamp/usr/local/bin/rpmsg-sample-ping-static
zynqmp_linux_r5_proc_init
metal: info:      metal_uio_dev_open: No IRQ for device 40000000.shm.
Successfully open shm device.
metal: info:      metal_uio_dev_open: No IRQ for device 40320000.shm.
Successfully open shm poll device.
Successfully added shared memory
Successfully initialized Linux r5 remoteproc.
Successfully initialized remoteproc
pa:0x40008e90
pa:40008e90
Calling mmap resource table.
Successfully mmap resource table.
Successfully set resource table to remoteproc.
before metal_io_phys_to_virt
Creating virtio...
Successfully created virtio device.
 1 - Send data to remote core, retrieve the echo and validate its integrity ..
RPMSG endpoint is binded with remote.
strlen(HELLO_MSG):15
rpmsg sample test: message 1 sent
rpmsg_endpoint_cb
received message from zephyr1: hello vflouris!  of size 16
 seed hello perister!:
 rnum 0:
rpmsg sample test: message 2 sent
rpmsg_endpoint_cb
received message from zephyr2: hello vflouris!  of size 16
 seed hello perister!:
 rnum 1:
rpmsg sample test: message 3 sent
rpmsg_endpoint_cb
received message from zephyr3: hello vflouris!  of size 16
 seed hello perister!:
 rnum 2:
rpmsg sample test: message 4 sent
rpmsg_endpoint_cb
received message from zephyr4: hello vflouris!  of size 16
 seed hello perister!:
 rnum 3:
rpmsg sample test: message 5 sent
rpmsg_endpoint_cb
received message from zephyr5: hello vflouris!  of size 16
 seed goodbye!:
 rnum 4:
**************************************
 Test Results: Error count = 0
**************************************
```

```
*** Booting Zephyr OS build v3.0.0-rc1-349-gfc9f976485c5  ***
Starting application thread!

OpenAMP[remote]  linux responder demo started
                                rsc_table:800040008e90
vring 0:800040010000
vring 1:800040014000
rpmsg_recv_callback
Message 0 received from linux:hello perister!
Message cleared  :
rpmsg send returns:16
rpmsg_recv_callback
Message 1 received from linux:hello perister!
Message cleared  :
rpmsg send returns:16
rpmsg_recv_callback
Message 2 received from linux:hello perister!
Message cleared  :
rpmsg send returns:16
rpmsg_recv_callback
Message 3 received from linux:hello perister!
Message cleared  :
rpmsg send returns:16
rpmsg_recv_callback
Message 4 received from linux:goodbye!rister!
Message cleared  :
rpmsg send returns:16
task_end!
```

Figure 18: Screenshot of OpenAMP ping pong experiment on Ariane-SDV between Linux (top-half) and Zephyr (bottom-half).

8.2 Evaluation of alternative PyFTI saving modes

Experiment setup

We have made resilience and performance evaluations using the Tensorflow framework. PyTorch support has been added recently, so we still do not have a proper evaluation using this framework.

In the first part of the experiments, we verify the correctness and reliability of our PyFTI module. That is, we make sure that PyFTI is functional and that its recovered execution has comparable accuracy to a normal execution, even under worst case scenarios. In order to do that, we perform two different experiments. The first one consists of the training of a MobileNet model with the cifar10 dataset. A total of 6 executions are performed, 3 normal executions and 3 executions using the PyFTI module. In order to account for the mentioned worst case scenario, the PyFTI executions will simulate a failure after every epoch, and recover from there afterwards. All executions train for a total of 15 epochs. A second experiment is added for completeness. The executions train a DenseNet121 model using the cifar100 dataset. A total of 3 executions are performed, one being a normal execution and the other two being worst case scenarios for PyFTI and Tensorflow checkpoint executions. Like in the previous experiment, the executions train for 15 epochs.

The first experiment is performed without any control of the randomness. That is, every execution starts with different random seeds and there is greater potential for variance in the results. On the other hand, the second experiment is performed with as much randomness control as Tensorflow allows us. This difference in experiment methodologies is applied so that we can observe the behavior of our PyFTI module in both equally relevant scenarios. As performance is not a factor when evaluating functionality and resilience, these experiments are performed on a local machine using Horovod with 4 MPI ranks.

The second part of the experiments is performed in the CTE-Power cluster from Marenostrum 4 Supercomputer. It is a cluster of 52 nodes, each node containing the following hardware:

- 2 x IBM Power9 8335-GTH @ 2.4GHz (3.0GHz on turbo, 20 cores and 4 threads/core, 160 threads per node)
- 512GB memory in 16 dimms x 32GB @ 2666MHz
- 2 x SSD 1.9TB as local storage
- 2 x 3.2TB NVME
- 4 x GPU NVIDIA V100 (Volta) with 16GB HBM2

All nodes have access to a general parallel file system (GPFS) via a fiber link of 10 Gbps, and all nodes are connected through an Infiniband interconnection network. For this experiment we use 32 ranks, with a total of 8 nodes and 4 ranks per node. Each rank has dedicated access to a GPU. The experiment consists in the training of a ResNet152 model with the cifar100 dataset during an execution without checkpointing, an execution using Tensorflow checkpoints and several other executions with different PyFTI saving modes. For standard Tensorflow checkpoints, we choose the GPFS as our storage medium, since it only supports a single checkpoint location. For PyFTI checkpoints, we choose the NVME drives as our local storage medium, and the GPFS for our global storage medium.

Evaluation

The results of the first resiliency experiment are shown in Figure 19. As we can observe, while each execution has its own unique learning progression due to the lack of randomness control, the accuracy ends up converging to similar values. Even the first PyFTI execution (*Recovery 1*), which seems to suffer from a slow start, has accuracy values very similar to other executions from epoch 10 onward.

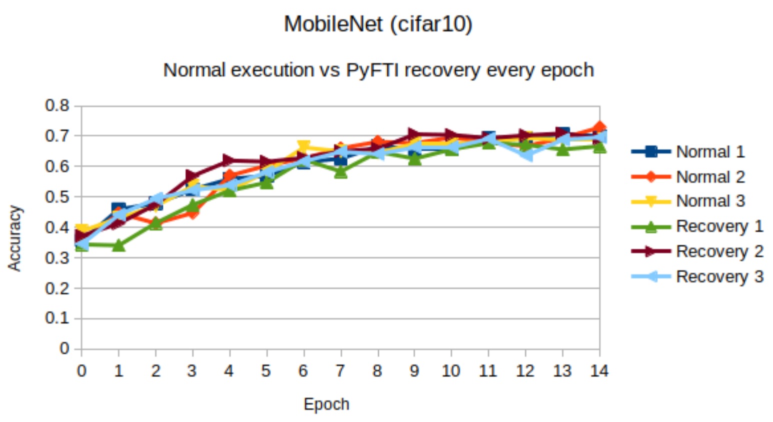


Figure 19: Comparison of a normal execution with PyFTI executions while recovering at every epoch, with no randomness control.

The results of the second experiment are shown in Figure 20. We can immediately notice a much smaller variance in comparison to the previous experiment, thanks to controlling the random seeds of the python modules. However, we were not able to provide deterministic restarts, so accuracy values still differ slightly. With these experiments, we can conclude that PyFTI provides strong resiliency to DL applications even in worst case scenarios, in which the applications fail frequently.

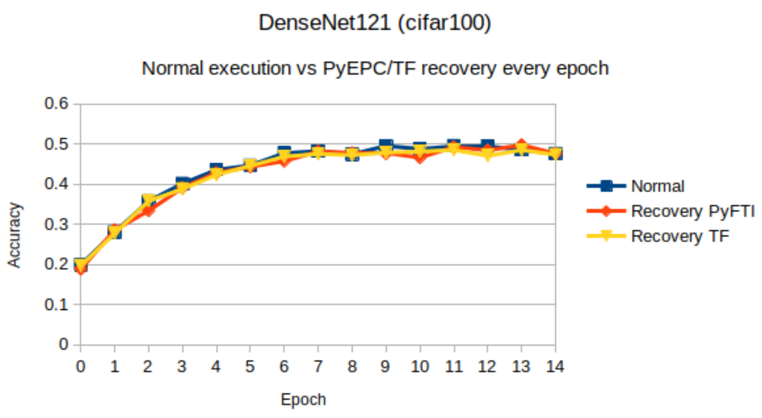


Figure 20: Comparison of a normal execution of PyFTI while recovering at every epoch, with randomness control.

The results of the performance evaluation, specifically overhead analysis, are shown in Figure 21. We can immediately notice that the **non-partitioning** saving mode generates a lot of overhead in comparison to any other execution. This is because this mode does not assume that the checkpoint data is the same for all ranks, and thus performs a complete serialization and storage of the data for each rank. While this should be the standard behavior in many applications, it is very inefficient in an environment where we know that all the checkpoint data is the same at the moment of performing a checkpoint. This shows the clear need for optimizations that reduce the checkpoint overhead in these scenarios, which are very common in machine learning applications.

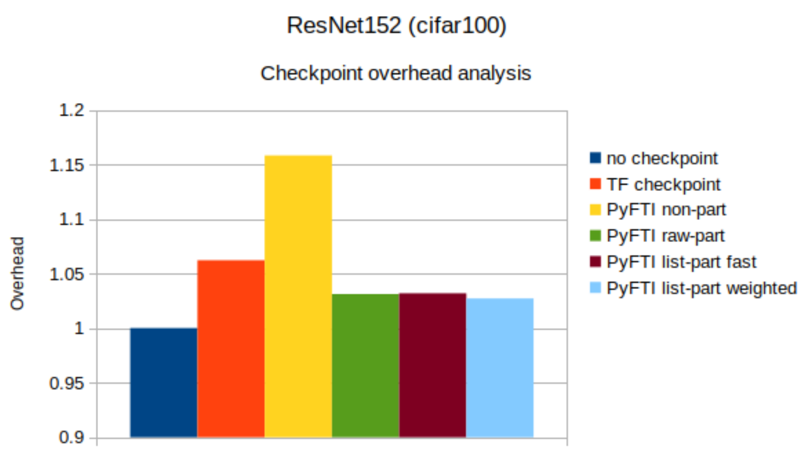


Figure 21: Checkpoint overhead analysis when training ResNet152 using cifar100 dataset.

We can see a substantial reduction of the overhead with the PyFTI partition-based saving modes, even compared to standard Tensorflow checkpoints. This is within our expectations, as the standard Tensorflow checkpoints are performed in a single storage location (in the GPFS), while PyFTI leverages NVME storage with reliability techniques in several checkpoint levels, only flushing data to the GPFS in the last checkpoint level. Furthermore, our checkpoint partitioning techniques allow for faster parallel checkpoint writes and the reduction of computational workloads assigned to some checkpoint levels.

Performance differences between **raw-partitioning**, **list-partitioning fast** and **list-partitioning weighted** modes are not as obvious. In fact, we argue that none of these modes are strictly better than others, but rather work better in certain scenarios. In our experiment, for example, the **list-partitioning fast** mode does not seem to provide any improvement over **raw-partitioning** mode. This is likely because, while it saves computation time in the serialization, it ends up with a poor checkpoint data distribution across ranks, worsening the I/O write performance. On the other hand, **list-partitioning weighted** mode handles the data distribution in a smarter way, so we can appreciate a slight overhead reduction. In summary, we can see overheads as low as 2.7%, 2.3x less overhead than standard Tensorflow checkpoints.

9. List of Abbreviations

AHB	Advanced High-performance Bus
AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
AMAT	Average Memory Access Time
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuit
ASID	Address Space IDentifier

ATPG	Automatic Test Pattern Generation
AXI	Advanced eXtensible Interface
BIST	Built-In Self-Test
BRAM	Block Random Access Memory
BSD	Berkeley Software Distribution
BSP	Board Support Package
BTB	Branch Target Buffer
C2C	Chip-to-Chip
CCI	Cache Coherent Interconnect
CDB	Common Data Bus
CFD	Computational Fluid Dynamics
CHI	Coherent Hub Interface
CI/CD	Continuous Integration / Continuous Delivery
CLIC	Core Local Interrupt Controller
CLINT	Core-Local INTerrupter
CMO	Cache Management Operation
CNN	Convolutional Neural Network
COM	Computer On-Module
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSR	Control and Status Register
CTG	Compatibility Test Generator
CUPs	Cell Updates Per Second
CV	Computer Vision
DAG	Direct Acyclic Graph
DDI	DNS/DHCP/IPAM
DDR	Double Data Rate
DFS	Dynamic Frequency Scaling

DFT	Discrete Fourier Transform
DFT	Design For Testing
DMA	Direct Memory Access
DMAR	Direct Memory Access Remapping
DNA	DeoxyriboNucleic Acid
DP	Double Precision (floating-point)
DRAM	Direct Random Access Memory
DRC	Design Rule Check
DSP	Digital Signal Processor
DV	Design and Verification
E-ATX	Extended Advanced Technologies eXtended
EAPI	Embedded Application Programming Interface
ECC	Error Correcting Codes
ED2P	Energy-Delay Square Product
EDP	Energy-Delay Product
ELF	Executable and Link Format
eMMc	embedded MultiMedia card
EPI	European Processor Initiative
FCBGA	Flip-Chip Ball Grid Array
FDSOI	Fully Depleted Silicon-On-Insulator
FF	Flip-Flop
FFT	Fast Fourier Transform
FIFO	First In First Out
FIT	Failures-In-Time
FLOPS	Floating-Point Operations Per Second
FMA	Fused Multiply-Add
FP	Floating-Point

FPGA	Field-Programmable Gate Array
FPS	Frames Per Second
FPU	Floating-Point Unit
FSM	Finite State Machine
gdb	GNU Debugger
GDSII	Graphic Design System II stream format
GLS	Gate Level Simulation
GPIO	General-Purpose Input/Output
GPU	Graphics Processing Unit
HAL	Hardware Abstraction Layer
HBM	High Bandwidth Memory
HLS	High-Level Synthesis
HMI	Human Machine Interface
HPC	High-Performance Computing
HPDA	High-Performance Data Analytics
HPM	Hardware Performance Monitoring
HPM	Hardware Performance Monitoring
I/O	Input/Output
I2C	Inter-Integrated Circuit
IC	Integrated Circuit
IoT	Internet of Things
IOVA	Input/Output Virtual Address
IP	Intellectual Property
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
ISG	Instruction Sequence Generator
KPI	Key Performance Indicator

LD/ST	Load/Store
LF/s	Left-First mappings per second
LR	Load Reserved
LRU	Least Recently Used
LSQ	Load/Store Queue
LT	Link Traversal
LUT	LookUp Table
LVCMOS	Low-Voltage Complementary Metal Oxide Semiconductor
LVDS	Low-Voltage Differential Signaling
LVS	Layout Versus Schematic
MAC	Multiply ACcumulate
MAC	Medium Access Control
ML	Machine Learning
MMU	Memory Management Unit
MPI	Message Passing Interface
MPKI	Misses Per Kilo Instruction
MPW	Multi Project Wafer
NIC	Network Interface Controller
NoC	Network on-Chip
NUCA	Non-Uniform Cache Access
OoO	Out-of-Order
OS	Operating System
P&R	Place and Route
PC	Program Counter
PCB	Process Control Block
PCIe	Peripheral Component Interconnect express
PDE	Partial Differential Equation

PE	Processing Element
PHY	PHYsical layer
PICMG	PCI Industrial Computer Manufacturers Group
PL	Programmable Logic
PLIC	Platform-Level Interrupt Controller
PLL	Phased Lock Loop
PMA	Physical Memory Attributes
PMC	Performance Monitoring Counter
PMP	Physical Memory Protection
PMP	Physical Memory Protection
PMU	Performance Monitoring Unit
PS	Processor System
PTE	Page Table Entry
PTW	Page Table Walker
QoS	Quality of Service
QSPI	Quad Serial Peripheral Interface
RAS	Return Address Stack
RAT	Register Alias Table
RDAM	Remote Direct Memory Access
RECS	Resource Efficient Computing and Storage
ReLU	Rectified Linear Unit
RNA	RiboNucleic Acid
ROM	Read-Only Memory
RTL	Register-Transfer Level
RU	Rack Units
RVV	RISC-V Vector Extension
RVWMO	RISC-V Weak Memory Ordering

S-NUCA	Static Non-Uniform Cache Access
SA	Switch Allocation
SATA	Serial Advanced Technology Attachment
SATP	Supervisor Address Translation and Protection
SC	Store Conditional
SDV	Software Development Vehicle
SECDED	Single Error Correction Double Error Detection
SerDes	Serializer/Deserializer
SIMD	Single Instruction Multiple Data
SO-DIMM	Small Outline Dual In-line Memory Module
SoC	System on-Chip
SPI	Serial Peripheral Interface
ST	Switch Traversal
STA	Static Timing Analysis
SV	System Verilog
SVA	System Verilog Assertions
TLB	Translation Lookahead Buffer
TSO	Total Store Order
UART	Universal Asynchronous Receiver Transmitter
UI	User Interface
ULL	Ultra-Low Leakage
UPF	Unified Power Format
URAM	Ultra Random Access Memory
USB	Universal Serial Bus
UVM	Universal Verification Methodology
VC	Virtual Channel
VIP	Verification Intellectual Property

VLEN	Vector LENgth
VLSQ	Vector Load/Store Queue
VPN	Virtual Page Number
VPU	Vector Processing Unit