On the Containerization and Orchestration of RISC-V architectures for Edge-Cloud computing

Francesco Lumpp
Dept. of Innovation Medicine - Univ. Verona
Italy, Verona
francesco.lumpp@univr.it

Francesco Barchi
Dept. of Electrical, Electronic, and Information Engineering - Univ. Bologna
Italy, Bologna
francesco.barchi@unibo.it

Andrea Acquaviva
Dept. of Electrical, Electronic, and Information Engineering - Univ. Bologna
Italy, Bologna
andrea.acquaviva@unibo.it

Nicola Bombieri
Dept. of Innovation Medicine - Univ. Verona
Italy, Verona
nicola.bombieri@univr.it

ABSTRACT

Containerization technologies, orchestration systems and open hardware architectures, such as RISC-V, are crucial as the foundation of open digital infrastructures for the computing continuum - the seamless distribution of data and computation across platforms with heterogeneous capabilities. However, it is unknown how containerization technologies and orchestration systems, such as Kubernetes, would impact performance in new architectures based on RISC-V. This work aims to address this question and introduces KubeEdge-V, an orchestration platform for RISC-V systems. We define the minimum components required to support the basic features of the containerization and orchestration platforms: network plug-ins, container runtimes and computational/networking requirements, and we evaluated KubeEdge-V performance on a prototype of a distributed computing system based on SiFive processors called Monte Cimone. Finally, the paper compares the performance of KubeEdge-V on SiFive processors with an equivalent system based on ARM architecture featuring the same power envelope.

ACM Reference Format:
Francesco Lumpp, Francesco Barchi, Andrea Acquaviva, and Nicola Bombieri. 2023. On the Containerization and Orchestration of RISC-V architectures for Edge-Cloud computing . In *3rd Eclipse Security, AI, Architecture and Modelling Conference on Cloud to Edge Continuum (ESAAM 2023), October 17, 2023, Ludwigsburg, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3624486.3624490

1 INTRODUCTION

In recent years, there has been considerable interest in open hardware architectures, which allow innovation in processors to take place faster and faster, covering the whole computing continuum spectrum from cloud to edge and low-power systems. RISC-V architectures play a dominant role in this context for both academic and industrial product designs [4, 11].

Unlocking the potential of these architectures in the context of the computing continuum is essential to open the way to future systems entirely based on open and novel hardware. Indeed, the European Commission is pushing the development of HPC, cloud and edge computing systems based on RISC-V [2], which is already gaining momentum in edge and microcontroller architectures. Consequently, it is of utmost importance to concurrently bring the related software ecosystem at the same technology readiness level to support this evolution. A great effort is already happening in this direction, as indicated by the growing software libraries and tools for RISC-V [3]. This effort is still missing regarding software infrastructures and tools for the computing continuum. A first step in this direction requires a preliminary evaluation and profiling of the containerization software on RISC-V processors and platforms to identify possible bottlenecks to be faced in next-generation architectures. However, RISC-V processors are not yet present in commercial distributed computing platforms. Consequently, no container software version has yet been ported or profiled on these systems.

This paper aims to fill this gap by porting and profiling an orchestration platform (Kubedge-V) to a RISC-V cluster prototype based on SiFive processors. The cluster, called Monte Cimone, was designed to open the way to the future edge computing systems based on RISC-V [5]. We identified the minimum components to support the basic features of container orchestration, the required network plugins (e.g., Flannel, EdgeMesh), and the container runtime (e.g., runc with CRI-O) to be employed for the target RISC-V architecture.

In its current implementation, Monte Cimone performance per Watt lies in the class of high performance edge computing platforms. For this reason, we considered a reference architecture made of a Kubernetes node running on a high-performance server (with an Intel Xeon processor) and KubeEdge-V running on Monte Cimone. This setting allowed us to profile the execution of KubeEdge-V on RISC-V to characterize its overheads on different benchmark

ESAAM 2023, October 17, 2023, Ludwigsburg, Germany

ACM ISBN 979-8-4007-0835-0/23/10.
https://doi.org/10.1145/3624486.3624490

suites (Phoronix, OSBench, IPC-Benchmark and stress-ng). In addition, we show the overhead introduced by the KubeEdge-V system on both execution time and memory usage. Using a set of benchmarks and scaling the number of used containers, we show how the system performs under different load levels, finally comparing it with an equivalent system based on an ARM architecture featuring the same power envelope. The results show that the solution is feasible and that the performance degradation caused by containerisation on RISC-V systems is comparable to the performance degradation observed on the ARM system. We also identify which operations are mostly impacted by the container runtime and should be considered for future software or hardware optimizations.

2 BACKGROUND AND RELATED WORK

This section introduces the RISC-V architecture and the importance of an open-source instruction set architecture. It also provides an analysis of the efforts made in the state-of-the-art research work to analyze containerization and orchestration overhead on performance, focusing on edge-based computing platforms.

2.1 RISC-V CPU

RISC-V is an open-source instruction set architecture (ISA) developed by researchers at the University of California, Berkeley, in 2010 [18]. The acronym RISC stands for Reduced Instruction Set Computer, which means the architecture has a smaller set of simple and standardized instructions. The simplicity and modularity of the RISC-V ISA make it ideal for various computing devices such as smartphones, tablets, embedded systems, but also high-performance computing (HPC) systems. Additionally, the open-source nature of RISC-V allows anyone to design and manufacture chips based on the architecture, encouraging innovation and competition in the market.

The RISC-V architecture has evolved over the years, and various versions have been released, including RV32I, RV32E and their 64 bits counterparts. The different letters in the naming scheme indicate variations in the ISA features. For example, “I” stands for the base integer instructions, “E” stands for the embedded profile, “F” includes instructions for single-precision floating-point arithmetic, and “D” for double-precision floating-point arithmetic. The RISC-V architecture also supports extensions that can be added to the base ISA to enhance functionality, such as the vector “V”, bit manipulation “B”, and compressed-instructions “C” extensions.

RISC-V technologies have gained popularity in recent years, with many companies adopting the architecture in their products. For instance, SiFive, a leading RISC-V chip designer, provides a range of processors for various applications, including embedded systems, IoT devices, and HPC systems. Other companies, such as Western Digital, NVIDIA, and Qualcomm, use RISC-V in their products.

The future of RISC-V technologies looks promising, with ongoing developments and collaborations among industry players and academia. The European Commission, for instance, in the context of the chip sovereignty strategy, is pushing open-source architectures and RISC-V, in particular, as a future seed for HPC systems [1]. This leads to a possible future scenario of edge and cloud systems composed of heterogeneous architectures with specific features (enabled by the RISC-V customizability), where computing continuum technologies play a crucial role in achieving global optimization thanks to orchestration.

2.2 Containerization and orchestration for Edge-Cloud architectures

Many works in literature have analyzed the impact of containerization, with several benchmarks emerging as standard, such as CPU compression/decompression of files, system memory and storage latency, as well as network bandwidth and latency. Other works have also analyzed specific applications such as MySQL for cloud environments and REST applications for IoT.

The authors at IBM have analyzed the impacts of containerization with many different benchmarks, such as `pxz` for decompression, `linpack` for floating point performance, `Stream` for memory and `netperf` for the network. They found little performance overhead and some network latency degradation due to the Network Address Translation (NAT) [10]. In an HPC-focused work [17], the authors have used different benchmarks, such as `sysbench`, `Stream` and `HPCG`, to measure the performance impact of containerization, finding no significant CPU overhead, but discovering a higher memory usage for containerized applications. In [16], the authors used the `Phoronix test suite` to benchmark the containerization overhead and analyze the impact of orchestration. They found minimal overhead in containerization and a worst-case scenario of 8% reduced performance when using an orchestration platform such as Kubernetes.

Internet-of-Things (IoT) architectures have also been analyzed with several different benchmarks. In [13], the authors used `7zip` for compression/decompression, `OpenSSL` for cryptography, `RAMspeed` for system memory latency, `Tio` for disk performance and `sockperf` for the network. In [9], the authors tested edge architectures based on AWS Greengrass and Microsoft Azure IoT Edge with custom-made benchmarks for speech and image recognition and found some limitations in system throughput when using containerization.

Other works, such as [15] and [12], have also compared the performance of containerization on different architectures like x86 and ARM. In the first work, many benchmarks were used, such as `lmbench` , `netperf`, `sysbench` and `linpack`, showing negligible overhead in all types of tests. In the second work, the authors analyzed the impact of containerization on REST services, finding that while the overhead of containerization is negligible, there are latency spikes when measuring end-to-end latency for the service.

When analyzing containers, the authors of [19], focused on the delay between a container being scheduled for execution and the container actually starting, and found that delay is very low and predictable, correlating the delay to the number of containers active on the device.

In [6], the authors built a complex architecture for data analysis, with edge nodes collecting data to compute and a suite of applications to analyze and store the data. These applications are distributed either on the edge nodes themselves or on fog or cloud nodes. They found that edge computing is a valid and strong alternative to fog or cloud computing while the amount of data to compute is lower than a threshold, thanks to a much lower average latency.

Figure 1: The Monte Cimone Server Blade hosts two SiFive Freedom U740 SoCs and has a form factor of 4.44 cm (1 Rack-Unit) in height, 42.50 cm in width, and 40 cm in depth. Each RISC-V board has dimensions of 170 mm by 170 mm. Image from [5].

Since no prior work on the impact of containerization and orchestration on RISC-V exists, we present such an analysis, which includes the impact and scaling, by using a series of synthetic benchmark suites: `sysbench`, `the Phoronix test suite`, `Stream`, `OSBench`, `the IPC_benchmark` and `stress-ng`.

3 THE KUBEEDGE-V PLATFORM

In this work, we developed and characterized an orchestration platform, KubeEdge-V, on a prototype RISC-V system. The prototype system is a computer cluster called "Monte Cimone". Monte Cimone nodes have been designed to explore RISC-V as the core for future edge nodes, as will be summarized in Section 3.1. Currently, the computational power delivered by these nodes makes them suitable for edge applications. For this reason, the reference system we consider is made of an x86 as the cloud node and RISC-V as the edge node, where we ported and profiled the KubeEdge platform, as explained in Section 3.2.

3.1 The Monte Cimone cluster

Monte Cimone represents a prototype and experimental platform of a comprehensive RISC-V (RV64) computing cluster, enclosing all the essential hardware components apart from processors, such as primary memory, non-volatile storage, and interconnect [5]. It is composed of eight computing nodes; each one is based on the U740 SoC from SiFive and integrates four U74 RV64GCB application cores (the U74 processor is a dual issue in-order execution pipeline, with a peak sustainable execution rate of two instructions per clock cycle), running up to 1.2 GHz and 16 GiB of DDR4, 1 TiB node-local NVME storage, and PCIe expansion cards. The U740 is Linux-capable SoC with a consumption of 6 Watt, placing the system in a low-power category, making it suitable for an edge-computing role.

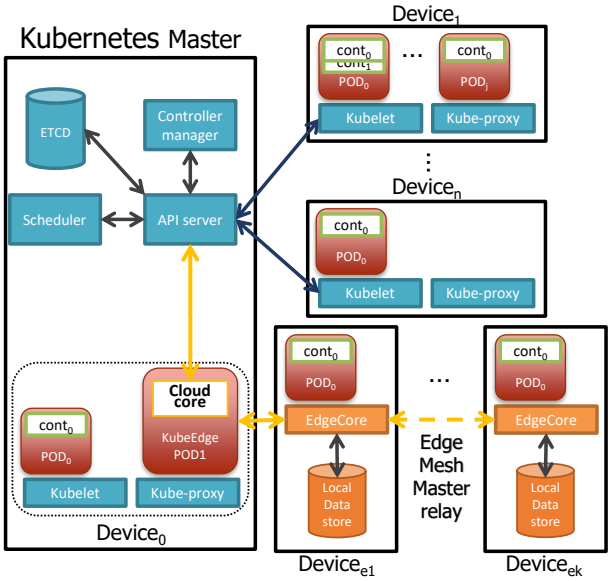

Figure 2: Kubernetes and KubeEdge architecture.

Actually, Monte Cimone nodes use a 1 Gbps Ethernet copper interface. As described in [5], to improve the network throughput and the communication latency, the authors tested a Mellanox ConnectX-4 FDR HCA (Infiniband FDR HCA (56Gbit/s) successfully ran an IB ping test and showed that a full InfiniBand support could be feasible. This feature is currently under development and not actually available due to incompatibilities between the software stack and the kernel driver.

As the target is to head towards high performance edge/fog computing, with a final goal to scale towards HPC systems based on RISC-V, an entire HPC software ecosystem and a complete system monitoring infrastructure has been ported to Monte Cimone. In particular, it runs a software stack composed of a job scheduler (SLURM), an LDAP server, the Spack package manager with compilers toolchains and scientific libraries, and a monitoring framework based on ExaMon. It is established that actual HPC applications can be executed on Monte Cimone [5].

3.2 Kubernetes for RISC-V

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications [7]. Its architecture, the computing cluster, is composed of at least one device: the Kubernetes *master*. Multiple devices can be connected to the master, with each of them having one *kubelet* software unit. When devices are connected to the master, they become nodes of the cluster. Each kubelet manages one or more *pods*. Pods are logical units used to cluster related containers to share resources. Each pod can have one or more containers, each containing one application. Fig. 2 shows a summary where each Kubernetes component is highlighted in blue, pods are dark red, and containers are green. The Kubernetes master controls the state of the cluster nodes through a controller manager, a database of the cluster information (etcd), and a scheduler unit. The scheduler manages container deployments across the cluster nodes. The functional units (i.e., master

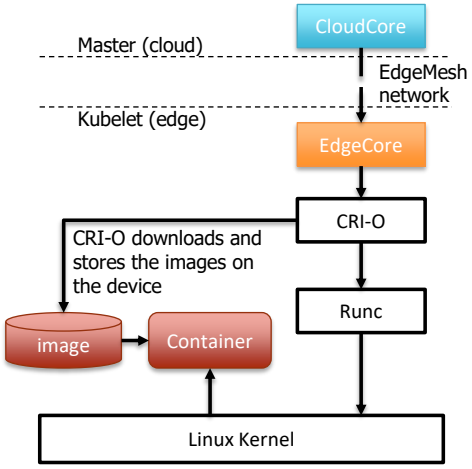

Figure 3: Software stack for KubeEdge on RISC-V.

and kubelets) communicate through the HTTP REST protocol. The master manages kubelet requests through an API server [8].

On the RISC-V edge nodes, we do not use Kubernetes but a lightweight, edge-oriented solution: KubeEdge. KubeEdge is an open-source system for extending native containerized application orchestration capabilities to hosts at the edge. It is built upon Kubernetes and provides fundamental infrastructure support for network, application deployment and metadata synchronization between cloud and edge [14]. Fig.2 shows the KubeEdge components in orange. KubeEdge is composed of two main components, the CloudCore, which is installed through a Kubernetes container on the master node, and the EdgeCore, which is a process that acts as a kubelet on the edge nodes. Unlike Kubernetes, KubeEdge can adapt to mesh networks and take into account nodes that momentarily become unreachable due to harsh conditions, e.g., remote sensors or drones. To achieve offline computing, KubeEdge maintains a copy of the cluster status locally on the edge device to then synchronize it with the master once connectivity is restored.

Fig. 3 shows a summary of the software stack required to run containers and KubeEdge on RISC-V. Container deployment requests are made through Kubernetes, which are then synchronized by the CloudCore to the appropriate EdgeCore, passing information through the mesh network. The EdgeCore on the target device, chosen by the Kubernetes scheduler, receives the deployment and requests a new container to CRI-O (Container Runtime Interface for the Open Container Initiative). If the container image is not already available on the device, CRI-O downloads it. CRI-O then starts *runc* (Container Runtime) with the requested parameters and network plugin (i.e., EdgeMesh). Runc starts the processes inside the container and assigns them to the appropriate namespace and cgroup through Linux system calls.

In this work, we compiled the whole environment for RISC-V, including the container images used to run Kubernetes. We also provide a detailed guide on Gitlab[1]. The installation process is summarized in the following.

[1]https://gitlab.com/parco-lab/kubeedge-v

3.2.1 Installing KubeEdge on RISC-V.. An up-to-date version of the Go language compiler is a main requirement to install the complete software stack. An older version of Golang can be obtained from the standard Ubuntu repository to bootstrap the latest. The installation process begins with runc and CRI-O. Configuration of CRI-O requires special attention, specifically adjusting the default container registry to a custom one for the pause image. The pause container is crucial for Linux to configure namespaces and cgroups before starting the actual container. However, since the pause image does not exist for RISC-V, it needs to be created and hosted on a custom registry. To achieve this, we extracted the Dockerfile from Kubernetes' pause and compiled it for RISC-V. Next, we require the EdgeMesh network plugin, which enables the communication between edge devices through a mesh network while relaying Kubernetes master data. The plugin requires a container running on each device for data handling and communication. Hence, we manually compiled EdgeMesh executables for both x86 and RISC-V, to create a single multi-architecture container. To support RISC-V, modifications were made to one of the libraries used by EdgeMesh. Detailed information is available on the Gitlab page. Finally, we compiled the KubeEdge executables, created the container images, and installed them. These operations involved extracting Dockerfiles from the original KubeEdge, as the original containers rely on special "builder" containers not yet available for RISC-V. With these steps completed, KubeEdge can be successfully installed and used.

4 EXPERIMENTAL RESULTS

In the following sections, we first explore the setup used to perform the benchmarks, such as hardware and software configurations. Then, we analyze the performance impact of containerization and orchestration on the RISC-V CPU performance and compare it with the overhead found on ARM. Finally, we analyze the impact of such a platform on the system memory of the RISC-V board.

4.1 Setup

We tested a set of benchmarks to quantify the containerization impact on the performance of the RISC-V architecture. We used *sysbench* for CPU integer performance, the *Stream* benchmark for system memory throughput, the *Phoronix test suite* for CPU-related benchmarks and finally, a sequence of system-related tests to verify the OS performance. We run each benchmark both natively and through KubeEdge. After benchmarking each test on the RISC-V board, we also verified the performance overhead on an ARM-based board, which allowed us to compare the results and verify if there was any odd behaviour on the new architecture.

To run the containerized benchmarks, we configured a Kubernetes cluster with the master running on an x86 device. Then, we connected two boards with KubeEdge. The first board is a SiFive HiFive Unmatched, part of the Monte Cimone cluster, and has a RISC-V architecture. It runs Ubuntu 21.10 with Linux Kernel 5.11 on 4 CPU cores running at 1GHz and 16GB of system memory. The second board is an ARM-based Jetson Xavier AGX running Ubuntu 20.04 with Linux kernel 5.10 on 8 CPU cores running at 2.3GHz and 16GB of system memory. We configured the Jetson to run with

only 4 cores at 1.2GHz, achieving a comparable CPU power target to the RISC-V board (i.e., $\approx 5W$).

In our analysis, we focused on a single node because our testing methodology is specifically designed to assess the architectural impact of containerisation, regardless of orchestration policies. Because of this, the network impact of orchestration has not been measured as it would not influence the outcome of this analysis. Regardless, it is important to note that the results obtained with the profiling we conducted are not restricted by the use of a single node and can be applied in broader scenarios.

4.2 Benchmark results

4.2.1 Sysbench. Table 1 shows the results obtained with sysbench, averaging 15 runs. The benchmark was run with 4 threads, calculating up to 1 million primes with a 1-hour time limit. The Unmatched board exhibits significantly lower performance compared to the Jeston, but it experiences less overhead from containerization.

4.2.2 Stream. Table 1 also shows the results for the STREAM benchmark run on the system memory, averaging 15 runs. The test was configured to run on 1.8GB of data with 4 threads. The memory subsystem of the Unmatched board is composed of 16GB of DDR4 memory running in dual channel with a 64bit bus at 1866MT/s, resulting in a theoretical maximum bandwidth of $\approx$ 30GB/s. The Jetson board has a much more sophisticated memory subsystem, composed of 16GB of LPDDR4x memory running in dual channel with a 256-bit bus at 1333MT/s, resulting in a theoretical maximum bandwidth of $\approx$ 86GB/s. As expected, due to these significant hardware differences, we observed that the RISC-V board is much slower in this test compared to the Jetson. On the other hand, the KubeEdge overhead is lower on the RISC-V architecture across all memory tests.

4.2.3 Phoronix. Table 2 shows the results obtained with the Phoronix test suite. We used the following benchmarks, and each run 15 times:

- Rodinia: this suite is focused on accelerator-based computing. We picked the LavaMD test based on OpenMP to benchmark multicore performance;
- x265: a CPU-based encoding test;
- 7-Zip: uses the integrated compression and decompression benchmarks;
- POV-Ray: Persistence of Vision Raytracer, it creates 3D graphics using ray tracing;
- OpenSSL: tests SSL (Secure Sockets Layer) and TLS (Transport Layer Security) protocols, including encryption and hashing functions.

This suite presents similar results to the other benchmarks. The Jetson is much faster, but the container overhead is, on average, 37.5% smaller on the RISC-V Unmatched board.

4.2.4 OSBench, IPC-Benchmark and stress-ng. Table 3 shows the results for OSBench, IPC-Benchmark and stress-ng, with each test run 15 times. OSBench performs a series of activities that are connected to running applications. The IPC benchmark tests the inter-process communication speed and bandwidth. Finally, stress-ng contains a multitude of tests, and those targeting the operating system were chosen.

The most interesting results are related to the file creation (i.e., the first two rows of Table 3) and process-related activities, i.e., launch programs, create processes, pthread and context switching. The former because it shows how the file system architecture of containers creates significant overhead in I/O-based operations and when excluded by mounting a native folder of the underlying file system inside the container, this overhead is not present anymore. This applies to RISC-V and to some extent to ARM64, where there is still some overhead even with the native mount. The latter shows how RISC-V incurs significant overhead when operations related to processes or threads are made. This is especially true when considering context switching, where RISC-V loses 21.4% of its performance.

To find the root cause of this delay on RISC-V, we run the `stress-ng` context switch benchmark again, along with the `perf` profiler. We found that system calls were taking significantly longer, up to 40% more time. This additional delay is probably caused by the container runtime system call interception procedure.

When a containerized process makes a system call, it is intercepted by the container runtime, which verifies permissions and resources before initiating the system call. To verify this theory, we formulated the following hypothesis: If the overhead is due to the container runtime rather than additional data structures from namespaces and cgroups, and if we manually associate a native process with the same namespaces and cgroups as a KubeEdge process, then:

(1) system call speed should be comparable to a native execution;
(2) cgroups/namespaces should work exactly like in KubeEdge.

To verify these hypotheses, we conducted the following experiment. We used two test applications: the first performs around 10 million system calls and calculates their average time, which should verify the initial hypothesis. The second test allocates 1GB of system memory using `malloc` with a cgroup limitation of 128MB, checking if the cgroup works properly by killing the application when it exceeds the limit. We used three configurations: native as a reference, containerized using KubeEdge, and manually associating the processes within namespaces/cgroups using `nsenter` and the `cgroup` parameter.

Table 4 shows the results. Native system calls and manual namespaces/cgroups have the same execution time, while there is a slowdown when executing them from KubeEdge. However, namespaces and cgroups were working correctly, as the memory allocation application was killed once it exceeded 128MB. Therefore, the overhead is definitely not caused by namespaces/cgroups, but it is reasonable to think that it is caused by the container runtime. This hypothesis does not justify the difference in overhead between RISC-V and ARM, but, as we manually ported the container runtime in this work, there may be differences due to implementation or architectural optimizations missing for RISC-V.

4.3 Memory footprint and scaling analysis

We also analyzed the memory usage of this lightweight virtualization technique to assess potential overhead beyond CPU performance. This involved testing applications running inside and outside containers, as well as the additional software overhead incurred by the system for containerization and orchestration.

Table 1: STREAM benchmark results with native and KubeEdge configurations on both RISC-V and ARM64. ▲ higher is better, ▼ lower is better. The results include the relative standard deviation.

Suite	Test	RISC-V			ARM64			Unit
		Native	KubeEdge	Overhead	Native	KubeEdge	Overhead	(▲▼)
Sysbench	CPU	2.47 ± 0.39%	2.46 ± 0.19%	-0.4%	6.40 ± 0.06%	6.17 ± 1.72%	-3.7%	event/s ▲
Stream	Copy	1 294 ± 0.28%	1 274 ± 2.19%	-1.5%	22 765 ± 0.51%	20 500 ± 0.15%	-10.0%	MiB/s ▲
	Scale	1 079 ± 0.50%	1 096 ± 1.01%	1.6%	23 929 ± 0.42%	22 229 ± 0.09%	-7.1%	MiB/s ▲
	Add	1 181 ± 0.20%	1 180 ± 0.89%	-0.1%	24 866 ± 0.10%	24 719 ± 1.46%	-0.6%	MiB/s ▲
	Triad	1 165 ± 0.18%	1 191 ± 1.94%	2.2%	24 499 ± 0.25%	25 044 ± 0.16%	2.2%	MiB/s ▲
			Average:	0.4%		*Average*:	-3.8%	

Table 2: Phoronix test suite results with native and KubeEdge configurations on both RISC-V and ARM64. ▲ higher is better, ▼ lower is better. The results include the relative standard deviation.

Suite	Test	RISC-V			ARM64			Unit
		Native	KubeEdge	Overhead	Native	KubeEdge	Overhead	(▲▼)
Rodinia	LavaMD	12 298 ± 1.05%	13 462 ± 0.41%	-8.7%	1 731 ± 2.41%	1 742 ± 2.05%	-0.6%	s ▼
x265 3.4 [1e-3]	Bos. 1080p	150 ± 0.00%	140 ± 0.00%	-6.7%	1 240 ± 0.00%	1 230 ± 0.24%	-0.8%	fps ▲
	Bos. 4K	30 ± 0.00%	30 ± 0.00%	0.0%	340 ± 0.00%	330 ± 0.00%	-2.9%	fps ▲
7-Zip	Comp.	1 782 ± 0.80%	1 805 ± 0.77%	1.3%	7 972 ± 2.63%	6 983 ± 3.60%	-12.4%	MIPS ▲
	Decomp.	3 433 ± 0.54%	3 430 ± 0.13%	-0.1%	5 921 ± 1.36%	5 309 ± 1.10%	-10.3%	MIPS ▲
POV-Ray	Trace	2 948 ± 1.33%	2 948 ± 1.79%	-0.0%	541 ± 2.38%	541 ± 2.38%	0.0%	s ▼
OpenSSL	SHA256	66.1 ± 0.38%	63.1 ± 5.89%	-4.7%	1 589.5 ± 2.45%	1 593.1 ± 0.69%	0.2%	MB/s ▲
	SHA512	92.7 ± 1.08%	90.4 ± 0.63%	-2.5%	398.5 ± 0.38%	387.1 ± 0.40%	-2.9%	MB/s ▲
	RSA4096_s	41 ± 0.00%	42 ± 0.73%	0.5%	155 ± 0.47%	147 ± 0.48%	-5.2%	sign/s ▲
	RSA4096_v	3 139 ± 0.28%	3 124 ± 0.69%	-0.5%	11 01 ± 0.01%7	10 532 ± 0.05%	-4.4%	verify/s ▲
	AES-128	65.0 ± 0.37%	64.1 ± 0.12%	-1.5%	4 964.8 ± 0.07%	4 866.4 ± 0.09%	-2.0%	MB/s ▲
	AES-256	53.9 ± 0.55%	53.2 ± 0.62%	-1.3%	3 677.2 ± 0.01%	4 027.0 ± 0.05%	9.5%	MB/s ▲
	ChaCha20	231.1 ± 0.27%	227.3 ± 0.09%	-1.7%	2 213.2 ± 0.01%	2 080.9 ± 0.08%	-6.0%	MB/s ▲
	Poly1305	168.0 ± 0.26%	165.7 ± 0.33%	-1.4%	1 497.4 ± 0.04%	1 415.9 ± 0.03%	-5.4%	MB/s ▲
			Average:	-1.9%		*Average*:	-3.1%	

Firstly, we measured the average memory usage of the runc, CRI-O, KubeEdge and EdgeMesh processes. All these applications are required to run containers. Table 5 shows the results. The highest impacts are caused by KubeEdge, EdgeMesh and CRI-O, which combined use almost 250MB of system memory. This overhead is substantial and could limit the applicability of such an orchestration system for edge applications.

The second test uses a varying number of identical containers to analyze what is the memory overhead for each container started. The containers only start the sleep process and thus use no memory or CPU. We obtain the available memory readings by accessing the `/proc/meminfo` variable. Table 6 shows the results. The overhead is measured at 1.51MB per container when using 4 containers, but it grows to 1.92MB per container when there are 64 containers deployed. These overheads can be attributed to runc, which was measured at 1.5MB per container in Table 5. Overall, the memory impact of 64 containers running is 122.52MB, which is quite significant.

The last test compares memory utilization and performance in the *sysbench* memory benchmark. It compares five configurations: native execution with one thread in total and one thread per CPU core, containerized execution with one container and either one thread in total or one thread per CPU core, and containerized execution with one container per CPU core.

Table 7 shows the results. The first two rows compare native and containerized execution with only one thread. The difference is negligible despite the containerized benchmark using slightly more memory. This may be caused by the nature of libraries in containers, requiring static linking and resulting in higher system memory usage. When comparing versions with four workers, performance is similar across configurations, but memory usage increases significantly with multiple containers due to the less efficient nature of running multiple identical copies compared to letting the benchmark handle four threads independently.

5 DISCUSSION AND FUTURE WORK

The results presented in Section 4 demonstrate that containerization on the RISC-V architecture is a viable approach that does not impose significant performance overhead. In comparison to ARM64,

Table 3: OSBench, IPC-Benchmark and stress-ng results with native and KubeEdge configurations, on both RISC-V and ARM64. ▲ higher is better, ▼ lower is better. The results include the relative standard deviation.

Suite	Test	RISC-V			ARM64			Unit
		Native	KubeEdge	Overhead	Native	KubeEdge	Overhead	(▲▼)
OSBench	Create Files	426 ± 5.48%	596 ± 4.32%	-28.4%	183 ± 2.28%	279 ± 4.71%	-34.3%	μs ▼
	↳ mount	-	425 ± 5.42%	0.2%	-	190 ± 3.64%	-3.7%	μs ▼
	C. Thread	197 ± 2.15%	212 ± 0.28%	-7.1%	136 ± 2.27%	171 ± 2.06%	-20.6%	μs ▼
	C. Processes	402 ± 2.37%	452 ± 2.47%	-10.9%	262 ± 1.59%	272 ± 1.05%	-3.7%	μs ▼
	Launch Prog.	618 ± 1.07%	673 ± 0.23%	-8.2%	924 ± 0.71%	818 ± 1.32%	13.0%	μs ▼
	Malloc	1 399 ± 0.43%	1 424 ± 1.19%	-1.8%	565 ± 1.49%	542 ± 0.23%	4.2%	μs ▼
IPC bench.	TCP Socket	117 705 ± 5.01%	112 507 ± 14.08%	-4.4%	137 263 ± 2.35%	136 135 ± 0.99%	-0.8%	msg/s ▲
	PIPE Un.	270 793 ± 0.36%	274 776 ± 1.96%	1.5%	154 689 ± 0.25%	152 937 ± 0.38%	-1.1%	msg/s ▲
	PIPE FIFO	269 931 ± 1.24%	266 265 ± 1.35%	-1.4%	155 666 ± 0.33%	157 966 ± 0.47%	1.5%	msg/s ▲
	Unix Socket	95 177 ± 2.17%	95 469 ± 0.94%	0.3%	90 560 ± 1.13%	92 266 ± 1.37%	1.9%	msg/s ▲
Stress-ng	Mutex	158 964 ± 2.16%	135 805 ± 0.52%	-14.6%	57 472 ± 4.30%	56 235 ± 2.49%	-2.2%	ops/s ▲
	Malloc	99 067 ± 0.17%	97 948 ± 0.98%	-1.1%	54 320 ± 1.16%	52 001 ± 0.22%	-4.3%	ops/s ▲
	Forking	1 851 ± 2.34%	2 020 ± 1.98%	9.1%	1 601 ± 3.07%	1 684 ± 12.59%	5.2%	ops/s ▲
	Pthread	2 690 ± 1.07%	2 340 ± 0.53%	-13.0%	3 633 ± 2.06%	3 473 ± 0.91%	-4.4%	ops/s ▲
	CPU cache	23 254 ± 4.07%	23 549 ± 8.91%	1.3%	182 042 ± 1.59%	177 345 ± 1.07%	-2.6%	ops/s ▲
	Semaphores	845 130 ± 2.05%	793 821 ± 2.37%	-6.1%	201 253 ± 9.87%	194 155 ± 5.54%	-3.5%	ops/s ▲
	Matrix Math	518 ± 0.22%	507 ± 0.17%	-2.1%	3 698 ± 0.15%	3 770 ± 0.05%	1.9%	ops/s ▲
	Vector Math	348 ± 0.33%	354 ± 0.02%	1.8%	6 484 ± 0.53%	7 084 ± 0.69%	9.3%	ops/s ▲
	Functions	2 294 ± 0.36%	2 249 ± 0.36%	-2.0%	18 766 ± 2.58%	16 004 ± 2.43%	-14.7%	ops/s ▲
	Cntx switch	84 110 ± 4.90%	66 082 ± 5.53%	-21.4%	47 990 ± 2.48%	47 865 ± 2.39%	-0.3%	ops/s ▲
			Average:	−5.4%		*Average*:	−2.9%	

Table 4: Overhead analysis for RISC-V system calls under KubeEdge.

	Native	Manual ns/cgroup	KubeEdge
Syscall time	192.18 ns	191.72 ns	206.48 ns
OOM kill	No	Yes	Yes

Table 5: Average memory usage for KubeEdge software stack. The results include the relative standard deviation.

Process	Avg.Memory [MiB]
KubeEdge	87.4 ± 2.10%
EdgeMesh	82.6 ± 3.10%
cri-o	76.6 ± 3.00%
runc	1.5 ± 5.30%

Table 6: Scaling overhead of containers.

Containers [#]	Available Memory [MiB]	Overhead [MiB]
0	15 827	-
4	15 821	1.4
16	15 804	1.4
32	15 775	1.6
64	15 704	1.9

Table 7: Results for running sysbench memory benchmark in different process, thread and containerization configurations.

Conf.	Result [MiB/s]	CPU [%]	Mem. [MiB]
Native - 1T	306.7	24.9%	1000.0
1 cont. - 1T	305.9	25.0%	1000.4
Native - 4T	1058.5	92.7%	1000.0
1 cont. - 4T	1065.6	88.6%	1000.1
4 cont. - 1T	1078.9	95.7%	1068.4

containerized applications on RISC-V experience less performance degradation on average, maintaining performance levels comparable to native execution.

However, the tests conducted on the operating system level reveal certain shortcomings in the software implementation of the platform on RISC-V. Figure 4 shows the average overhead of the benchmarks of Table 2 as *application-based* and the average overhead of the benchmarks of Table 3 as *OS-based*. In the graph it is clear that there is a significant difference in the overhead depending on the class of benchmark used. Notably, the context switching test in Table 3 highlights a performance loss of 21.43% for RISC-V when containerized, in contrast to a mere 0.26% for ARM64. These delays can likely be attributed to a lack of optimization in the software stack specifically designed for container environments.

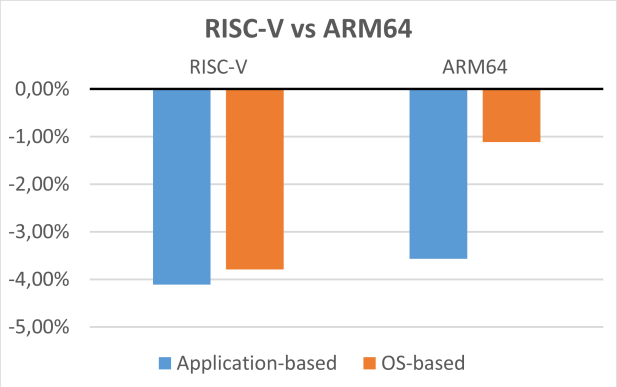

Figure 4: Graph comparing the performance loss for RISC-V and ARM64 in performance- and OS-related benchmarks.

Furthermore, it is worth noting that the memory overhead introduced by containerized applications running on RISC-V is substantial. With an additional system memory usage of nearly 250MB, certain edge computing devices, such as embedded boards with lower power targets and limited system memory, may lack the necessary capabilities to accommodate this complex orchestration platform.

Nevertheless, these findings underline the viability of containerization and orchestration on the RISC-V architecture, exhibiting similar benefits and limitations to those observed on ARM64. The deployment of containerized applications on RISC-V can generally be done without significant concerns regarding performance degradation. However, caution should be exercised when deploying system call-intensive applications or when utilizing this architecture on memory-constrained devices.

A future research goal would be analyzing the potential of distributed applications that have to take into account the orchestration and communication aspects of computation, which are unexplored within a containerised RISC-V-based system. Such an analysis of distributed applications, leveraging the capabilities of orchestration, would be needed to clarify the scalability and applicability of orchestration within the RISC-V architecture. This exploration could contribute to improving the suitability of RISC-V for high performance edge/fog computing, and eventually HPC systems.

6 CONCLUSIONS

This work effectively integrated the essential components of containerization and orchestration, such as network plugins and container runtime. The results demonstrate that these technologies can be efficiently implemented on a distributed computing system based on the RISC-V architecture with a small performance degradation; it is comparable to the performance degradation observed on the ARM architecture.

The experimental findings presented in this work carry significant implications for the development of open digital infrastructures. They broaden the possibilities for creating large computing ecosystems with an open and scalable infrastructure, leveraging the open hardware design of RISC-V as well as the adaptability and effectiveness of containerization and orchestration technologies.

ACKNOWLEDGMENTS

This study was carried out within the PNRR research activities of the consortium iNEST (Interconnected North-Est Innovation Ecosystem) funded by the European Union Next-GenerationEU (Piano Nazionale di Ripresa e Resilienza (PNRR) – Missione 4 Componente 2, Investimento 1.5 – D.D. 1058 23/06/2022, ECS_00000043). This manuscript reflects only the Authors' views and opinions, neither the European Union nor the European Commission can be considered responsible for them.

REFERENCES

[1] [n. d.]. *EU roadmap open hardware.* https://digital-strategy.ec.europa.eu/en/library/recommendations-and-roadmap-european-sovereignty-open-source-hardware-software-and-risc-v
[2] [n. d.]. *EuroHPC Risc-V.* https://eurohpc-ju.europa.eu/new-call-developing-hpc-ecosystem-based-risc-v-2023-02-01_en
[3] [n. d.]. *RISC-V Wiki.* https://wiki.riscv.org/display/HOME/RISC-V+Software+Ecosystem
[4] [n. d.]. *SiFive Website.* https://www.sifive.com/
[5] Andrea Bartolini, Federico Ficarelli, Emanuele Parisi, Francesco Beneventi, Francesco Barchi, Daniele Gregori, Fabrizio Magugliani, Marco Cicala, Cosimo Gianfreda, Daniele Cesarini, et al. 2022. Monte Cimone: Paving the Road for the First Generation of RISC-V High-Performance Computers. In *2022 IEEE 35th International System-on-Chip Conference (SOCC)*. IEEE, 1–6.
[6] Francisco Carpio, Marta Delgado, and Admela Jukan. 2020. Engineering and Experimentally Benchmarking a Container-based Edge Computing System. In *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*. 1–6. https://doi.org/10.1109/ICC40277.2020.9148636
[7] Cloud Native Computing Foundation. 2023. *Kubernetes.* https://kubernetes.io
[8] Cloud Native Computing Foundation. 2023. *Kubernetes Architecture.* https://kubernetes.io/docs/concepts/architecture/cloud-controller
[9] Anirban Das, Stacy Patterson, and Mike Wittie. 2018. EdgeBench: Benchmarking Edge Computing Platforms. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. 175–180. https://doi.org/10.1109/UCC-Companion.2018.00053
[10] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. 2015. An updated performance comparison of virtual machines and Linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 171–172. https://doi.org/10.1109/ISPASS.2015.7095802
[11] Eric Flamand, Davide Rossi, Francesco Conti, Igor Loi, Antonio Pullini, Florent Rotenberg, and Luca Benini. 2018. GAP-8: A RISC-V SoC for AI at the Edge of the IoT. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 1–4.
[12] Tom Goethals, Merlijn Sebrechts, Mays Al-Naday, Bruno Volckaert, and Filip De Turck. 2022. A Functional and Performance Benchmark of Lightweight Virtualization Platforms for Edge Computing. In *2022 IEEE International Conference on Edge Computing and Communications (EDGE)*. 60–68. https://doi.org/10.1109/EDGE55608.2022.00020
[13] Yinluo Jing, Zhiyi Qiao, and Richard O. Sinnott. 2022. Benchmarking Container Technologies For IoT Environments. In *2022 Seventh International Conference on Fog and Mobile Edge Computing (FMEC)*. 1–8. https://doi.org/10.1109/FMEC57183.2022.10062773
[14] KubeEdge. 2023. *KubeEdge.* https://kubeedge.io
[15] Vivian Noronha, Ekkehard Lang, Maximilian Riegel, and Thomas Bauschert. 2018. Performance Evaluation of Container Based Virtualization on Embedded Microprocessors. In *2018 30th International Teletraffic Congress (ITC 30)*, Vol. 01. 79–84. https://doi.org/10.1109/ITC30.2018.00019
[16] Yao Pan, Ian Chen, Francisco Brasileiro, Glenn Jayaputera, and Richard Sinnott. 2019. A Performance Comparison of Cloud-Based Container Orchestration Tools. In *2019 IEEE International Conference on Big Knowledge (ICBK)*. 191–198. https://doi.org/10.1109/ICBK.2019.00033
[17] Alfred Torrez, Timothy Randles, and Reid Priedhorsky. 2019. HPC Container Runtimes have Minimal or No Performance Impact. In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. 37–42. https://doi.org/10.1109/CANOPIE-HPC49598.2019.00010
[18] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. 2011. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62* 116 (2011).
[19] Tianming Wei, Madhav Malhotra, Bing Gao, Tomas Bednar, Derek Jacoby, and Yvonne Coady. 2017. No such thing as a "free launch"? Systematic benchmarking of containers. In *2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. https://doi.org/10.1109/PACRIM.2017.8121922