SCALABILITY 2024

The First International Conference on Systems Scalability and Expandability

ISBN: 978-1-68558-216-6

November 17th – 21st, 2024

Valencia, Spain

SCALABILITY 2024 Editors

Petre Dini, IARIA, USA/EU

SCALABILITY 2024

Forward

The First International Conference on Systems Scalability and Expandability (SCALABILITY 2024), held on November 17-21, 2024 in Valencia, Spain, inaugurates a series of events covering all aspects related to scalability challenges and solutions from design, to monitoring, and maintenance of computational systems.

The true definition of scalability has to do with meeting demand. Scalable design is a form of responsiveness. Scalability is the ability of a system to provide throughput in proportion to, and limited only by, available hardware resources. A scalable system is one that can handle increasing numbers of requests without adversely affecting response time and throughput.

The growth of computational power within one operating environment is called vertical scaling. Horizontal scaling is leveraging multiple systems to work together in parallel on a common problem. Cloud scalability in cloud computing refers to the ability to increase or decrease IT resources as needed to meet changing demand. Scalability is one of the hallmarks of the cloud and the primary driver of its exploding popularity with businesses. Scaling is one of the most important components of cloud cost management: performance vs. availability vs. scalability tradeoff is essential.

Data storage capacity, processing power and networking can all be scaled using existing cloud computing infrastructure. Better yet, scaling can be done quickly and easily, typically with little to no disruption or down time. Third-party cloud providers have all the infrastructure already in place; in the past, when scaling with on-premises physical infrastructure, the process could take weeks or even months and required tremendous expense.

Whether traffic or workload demands increase suddenly or gradually over time, a scalable cloud solution enables organizations to respond appropriately and cost-effectively to an increased need for storage and performance. The switch to cloud has improved the computing power for organizations that used to run servers on premises. The leading cloud providers - Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform - offer flexibility for organizations that may need to add or reduce resources at a moment’s notice.

Virtualization is what makes cloud scalability possible. In cloud computing, scaling is the process of adding or removing computing, storage, and network services to meet the demands a workload has for resources in order to maintain availability and performance as utilization increases.

The number of IoT devices might soon reach 39 billion. Other more complex systems (such as vehicles and power grids, equipped with sensing and storage capabilities) generate huge amounts of data of various types. 5G operation uses a large scale of heterogeneous IoT devices while being performance-efficient in real-time is challenging considering central processing by services hosted on geographically distant clouds (latency incurred and the ingress bandwidth).

To leverage resources located at the edge of the network forming a continuum between the cloud and the edge, the Fog/Edge computing paradigm is expected to increase not only scalability, but also the agile adaptation of sudden traffic, speed, and load changes. The approach deals with latency and bandwidth in close vicinity with data producers (IoT devices, such as home routers, gateways, or more substantial micro data centers) by harnessing the edge. This approach raises security aspects (trust, privacy, guarantees) in edge-based computing.

This conference was very competitive in its selection process and very well perceived by the international community. As such, it attracted excellent contributions and active participation from all over the world. We were very pleased to receive a large amount of top quality contributions.

We take here the opportunity to warmly thank all the members of the SCALABILITY 2024 technical program committee as well as the numerous reviewers. The creation of such a broad and high quality conference program would not have been possible without their involvement. We also kindly thank all the authors that dedicated much of their time and efforts to contribute to the SCALABILITY 2024. We truly believe that thanks to all these efforts, the final conference program consists of top quality contributions.

This event could also not have been a reality without the support of many individuals, organizations and sponsors. We also gratefully thank the members of the SCALABILITY 2024 organizing committee for their help in handling the logistics and for their work that is making this professional meeting a success.

We hope the SCALABILITY 2024 was a successful international forum for the exchange of ideas and results between academia and industry and to promote further progress in scalability and expandability research. We also hope that Valencia provided a pleasant environment during the conference and everyone saved some time for exploring this beautiful city

SCALABILITY 2024 Steering Committee

Xiaohua Feng, University of Bedfordshire, UK
Tzung-Pei Hong, National University of Kaohsiung, Taiwan
Byungchul Tak, Kyungpook National University, South Korea
Julian Kunkel, Georg-August-Universität Göttingen, Germany

SCALABILITY 2024 Publicity Chair

Francisco Javier Díaz Blasco, Universitat Politecnica de Valencia, Spain
Ali Ahmad, Universitat Politecnica de Valencia, Spain

SCALABILITY 2024

Committee

Table of Contents

Total Cost of Ownership: Cloud-based vs. Onboard Vehicle Software Components

Daniel Baumann, Martin Sommer and Eric Sax
Institut fuer Technik der Informationsverarbeitung (ITIV)
Karlsruhe Institute of Technology (KIT)
Engesserstr. 5, 76131 Karlsruhe, Germany
E-Mail: {daniel.baumann, ma.sommer, eric.sax}@kit.edu

Falk Dettinger and Michael Weyrich
Institute of Industrial Automation and Software (IAS)
University of Stuttgart
Pfaffenwaldring 47, 70550 Stuttgart, Germany
E-Mail: {falk.dettinger, michael.weyrich}@ias.uni-stuttgart.de

***Abstract*—The automotive industry is increasingly focusing on connected vehicles that have the opportunity to connect to external platforms, such as the cloud or edge. In this context, the electric/electronic (E/E) architecture is evolving from a signal-oriented to a service-oriented architecture where loosely coupled services, representing functions or the software components (SWCs) they are composed of, can be dynamically connected. At the same time, realization by means of independent services enables the execution both in the vehicle and in the communication network, like the cloud. The costs involved in developing, operating, and maintaining vehicle SWCs have a significant impact on whether it makes sense to execute them in the cloud. In this paper, the authors propose an approach to calculate the Total Cost of Ownership (TCO) with Capital Expenditures (CapEx) and Operating Expenses (OpEx) of SWCs for the two different execution platforms, vehicle and cloud. The TCO model includes the lifecycle of the function from development to usage and maintenance. In a case study with a machine learning SWC for the heating, ventilation and air conditioning (HVAC) function, the model is investigated and break-even periods for the two platforms are calculated.**

Keywords-Total Cost of Ownership; Electric/Electronic Architecture; Cloud-based Software Components; Cloud Computing.

I. INTRODUCTION

The automotive industry is rapidly moving towards connected vehicles, integrating a growing number of software-defined functions [1]. This transition presents a critical decision for upcoming Electric/Electronic (E/E) architectures: where to execute these functions or the Software Component (SWC) they are composed of, aboard the vehicle or in the cloud.

Safety-critical functions may require local processing to minimize latency and ensure robustness, while non-safety critical comfort functions are possibly suitable for cloud execution. Previous work identified cost as a significant factor in determining the suitability of the different execution platforms [2].

Therefore, a comprehensive cost analysis based on Total Cost of Ownership (TCO) is needed to enable high-quality decision-making for the cloud offloading. This paper contributes to the ongoing research on connected vehicle architectures by providing a structured model for evaluating the economic feasibility of cloud-based execution. We anticipate that this model will be valuable for both automotive E/E architects and software developers involved in the design and deployment of software-defined functions for the future of connected vehicles.

The remainder of the paper is structured as follows: Section II will describe the theoretical background necessary for understanding the models utilized in the paper. Section III will cover the related state of work, providing an overview of existing research and literature on existing TCO models. Following in Section IV, these models for the onboard and the cloud-based SWCs are described. The TCO model will then be analyzed with the two execution platforms cloud and onboard in a case study with a fleet size of 1000 and 5000 vehicles in section V. A discussion of the use cases and the cost reduction options will be provided in section VI. Finally, section VII presents a conclusion to the paper and outlines future work.

II. BACKGROUND

A. E/E architecture

The E/E architecture refers to the electrical and electronic system of a vehicle, which includes all electrical components and control units required to operate the vehicle [3].

Historically, specific functions were executed on dedicated Electronic Control Units (ECUs) with limited interconnectivity. This so-called distributed architecture has been replaced by the domain-oriented architecture, where software is abstracted from the hardware and logically subdivided according to functions instead of according to individual control units [4]. This means that more than one SWC runs on a single ECU. In the future, centralized architectures will increasingly be used. Fewer but more powerful control devices designated to High Performance Computer (HPC) run the software components that underlie the functionality [5].

Today's automotive applications use signal-oriented architectures in which the software and hardware components involved are closely coupled with each other. In order to manage the increasing proportion of software in the vehicle and to enable dynamic E/E architectures, so-called service-orientated architectures are being introduced. The various functions are designed as independent services that can interact with each other in a modular fashion. Instead of linking individual components directly with each other as in the signal-orientated architecture, the components are viewed as independent services that can communicate via defined interfaces. [6]

Service-oriented E/E architectures can be used to design vehicle systems in such a way that they can communicate seamlessly with cloud services. This enables the utilization of cloud resources, such as the provision of data-intensive services. [7]

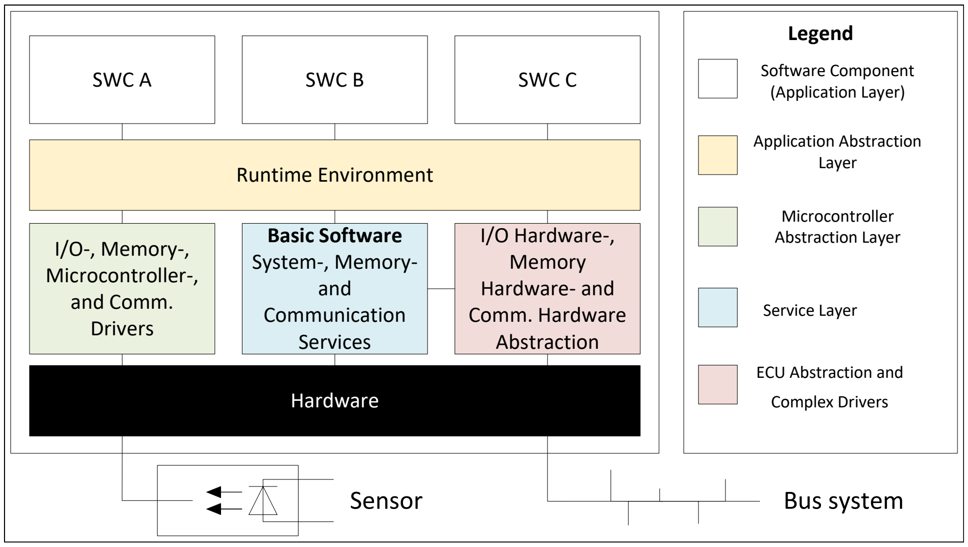
SWC A
SWC B
SWC C
Runtime Environment
I/O-, Memory-, Microcontroller-, and Comm. Drivers
Basic Software System-, Memory- and Communication Services
I/O Hardware-, Memory Hardware- and Comm. Hardware Abstraction
Hardware
Sensor
Bus system
Legend
Software Component (Application Layer)
Application Abstraction Layer
Microcontroller Abstraction Layer
Service Layer
ECU Abstraction and Complex Drivers

Figure 1. Software components within the AUTOSAR layered software architecture (adopted from [3]).

We define cloud-based SWC as follows, referring to and adapting Milani's definition [8]: "Cloud-based software components are regulation, control, or monitoring tasks that use the computing & storage capacities of the cloud instead of the available computing capacities of the vehicle. They can use both information from the vehicle and data from the cloud as input. The output of the relocated components optimize existing functions in the vehicle, replace them, or create a new function for themselves." The localization of software components can be explained using the layer structure of the AUTomotive Open System ARchitecture (AUTOSAR) architecture and refers to the application layer (s. Figure 1). Whereas a software component itself is defined as an "entity with discrete structure, such as an assembly or software module, within a system considered at a particular level of analysis" [9].

B. Cloud Computing

In general, Cloud Computing refers to the characteristics of a flexible and scalable infrastructure that conveys the illusion of unlimited, on-demand access to IT resources. The most widely adopted definition originates from the U.S. National Institute of Standards and Technology (NIST) [10]: "Cloud computing is a model to enable ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." In cloud computing, three different service models have emerged for accessing cloud resources: Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) [10].

C. Total cost of ownership

TCO is a financial estimate of the overall cost of a product or service over its entire lifespan, not just the initial purchase price. TCO consider all the direct and indirect costs associated with owning and using the product or service. First introduced by Ellram and Siferd in 1995, the concept of TCO has become widely adopted across industries and academia as a means of evaluating the long-term economic viability of investments. [11] TCO encompasses both the initial Capital Expenditures (CapEx) and the aggregate of Operating Expenses (OpEx) [12]:

$$TCO = CapEx + OpEx \tag{1}$$

CapEx refers to the upfront costs incurred at the time of purchasing the product or service. In contrast, OpEx are the ongoing costs associated with owning and using the product or service over its entire lifecycle.

III. Related Work

In their paper, Martens et al. [13] introduce a TCO approach tailored to cloud computing services. It outlines different pricing structures for cloud computing services and develops a formal mathematical model. The cost categories identified include strategic decision-making, evaluation and selection of service providers, service charges for different cloud models, implementation, support, initial and ongoing training, maintenance and modification, system failures, and backsourcing. A case study is presented as an IaaS example, detailing the cost types and related cost factors.

Kashef et al. [14] provide a detailed specification of cloud computing costs for hybrid clouds. Twenty cost factors are identified in the categories electricity, hardware, software, labor, business premises and cloud service. Costs are broken down into fixed and variable costs over time. The costs of a scenario for running an in-house data center with ten different services is shown.

Walterbuch et al. [15] introduce a TCO model tailored for cloud computing services in a public cloud environment. The example demonstrates the provisioning of a public IaaS Cloud Computing Service.

The paper by Heinrich et al. [16] proposes a TCO model for cloud computing, covering the cost of adoption, procurement, migration, operation (external and internal), usage, and exit. A case study is presented, comparing two scenarios: a Serverless Scenario and a Lift and Shift Scenario, against an on-premises architecture. Serverless refers to a platform for deploying applications without the user having to care about the underlying infrastructure. In contrast, the term lift and shift is used to describe a function that is migrated to the cloud. The study shows that the operation cost for cloud computing is lower than that on-premises. In terms of total cost, the lift and shift scenario is less expensive than on-premises after 15 years. The serverless scenario is always more expensive than the on-premises solution.

There is a gap in the literature regarding the comparison of traditional onboard functions with cloud computing services, especially within the automotive sector. This paper aims to address this gap by providing a detailed analysis of the TCO of the automotive onboard and cloud functions. Furthermore, we identify and analyze cloud computing as an enabler for further automotive functions and services.

IV. TCO Model for software components

The development and operation of vehicle SWC in the cloud or on board the vehicle is associated with costs. The TCO of a SWC TCO_{SWC} consist of the development costs C_{dev}, the deployment costs C_{depl}, both capital expenditures, and execution costs C_{exe} which are operating expenses on a monthly basis:

$$TCO_{SWC} = \underbrace{C_{dev} + C_{depl}}_{CapEx} + \underbrace{C_{exe}}_{OpEx} \tag{2}$$

Expert interviews were conducted with a German Original Equipment Manufacturer (OEM) for the creation and evaluation of the TCO model. The cost components of the onboard function and the cloud function are explained in more detail below.

The following cost breakdown relates to the SWC shown in Figure 1, which are assigned to the application layer.

A. Onboard SWC

Similar to equation 2 the costs of an onboard vehicle SWC can be summarized as:

$$TCO_{v,SWC} = C_{v,dev} + C_{v,depl} + C_{v,exe} \tag{3}$$

1) Development: The costs of developing an onboard vehicle SWC $C_{v,dev}$ can be divided into software $C_{v,dev,sw}$ and hardware $C_{v,dev,hw}$. In the following, hardware always refers to ECUs with a microcontroller and the associated peripherals [17]:

$$C_{v,dev} = \sum_{i=0}^{n} C_{v,dev,sw} + \sum_{i=0}^{n} C_{v,dev,hw} \tag{4}$$

Software development costs $C_{v,dev,sw}$ include expenses related to designing, implementing, integrating and testing software functions used in vehicle control systems. The development costs of software increase due to the requirements for reliability and safety. This is because careful validation and verification are necessary to ensure that the software is error-free and robust. Hardware development costs $C_{v,dev,hw}$ include expenses for designing ECUs, sensors, actuators, and other physical components required for the function's functionality. These components must often meet strict requirements for robustness, reliability, and performance to withstand the demanding environmental conditions of vehicle operation.

2) Deployment: The deployment costs of an onboard vehicle function $C_{v,depl}$ consist of the sum of expenses associated with the material, production and logistic of the ECUs $C_{v,depl,ecu}$:

$$C_{v,depl} = \sum_{i=0}^{n} C_{v,depl,ecu} \tag{5}$$

3) Execution: The execution costs $C_{v,exe}$ pertain to ongoing expenses associated with the operation and utilization of onboard vehicle SWC. These costs encompass various aspects $C_{v,exe,n}$, including Over-the-Air (OTA) updates of ECUs, operation costs like the energy consumption costs, maintenance and repair costs and costs for customer support and service.

$$C_{v,exe} = \sum_{i=0}^{n} C_{v,exe,n} \tag{6}$$

B. Cloud-based SWC

As described in Section IV in equation 2, the cost of a cloud function can be calculated as follows:

$$TCO_{c,SWC} = C_{c,dev} + C_{c,depl} + C_{c,exe} \tag{7}$$

1) Development: The development of a cloud-based function $C_{c,dev}$ is associated with various costs, including internal introduction $C_{c,dev,intro}$, purchasing $C_{c,dev,pur}$, migration $C_{c,dev,mig}$ and software development $C_{c,dev,sw}$:

$$\begin{aligned} C_{c,dev} = &\sum_{i=0}^{n} C_{c,dev,intro} + \sum_{i=0}^{n} C_{c,dev,pur} \\ &+ \sum_{i=0}^{n} C_{c,dev,mig} + \sum_{i=0}^{n} C_{c,dev,sw} \end{aligned} \tag{8}$$

The internal introduction costs $C_{c,dev,intro}$ encompass strategic planning, training and ensuring security protocols. Strategic planning involves defining the objectives, scope and methods for transitioning from traditional onboard functions to the cloud. Training programs are essential to introduce employees with the new technology and ensure smooth introduction and use. Furthermore, it is essential to integrate security measures to protect data and systems from potential threats. This often requires investment in cybersecurity infrastructure. Procurement costs $C_{c,dev,pur}$ include the cost of purchasing or licensing the software, as well as any associated infrastructure costs. To run a function in the cloud, it is possible to migrate an existing function or develop a function from scratch. The other cost factor can be set to zero accordingly. Migration costs $C_{c,dev,mig}$ refer to the expenses associated with transitioning from existing systems to cloud-based infrastructure, including infrastructure switching, implementation, testing and configuration. Software development costs, on the other hand, relate to the implementation and testing phases of building cloud-based functions. The development costs $C_{c,dev,sw/hw}$ are divided between software and hardware, as with the onboard function, whereby the hardware here is limited to sensors, actuators and other physical components. In contrast to the onboard function, no execution platform needs to be developed.

2) Deployment: Capital expenditures for the deployment of SWCs in the cloud are regarded as very low and therefore negligible. The costs associated with deployment in the vehicle are costs listed in the execution costs for the cloud case. Shifting a SWC to the cloud, converts initial investments for computing power and storage requirements into OpEx.

$$C_{c,depl} = 0 \tag{9}$$

A touchscreen in the vehicle is required to use the function.

3) Execution: The execution of a cloud-based SWC is associated with various costs $C_{c,exe}$. These include the costs for the cloud service provider $C_{c,exe,CSP}$, the internal operation $C_{c,exe,int}$, the usage $C_{c,exe,use}$ and the update & upgrade of the function $C_{c,exe,u\&u}$:

$$\begin{aligned} C_{c,exe} = &\sum_{i=0}^{n} C_{c,exe,CSP} + \sum_{i=0}^{n} C_{c,exe,int} \\ &+ \sum_{i=0}^{n} C_{c,exe,use} + \sum_{i=0}^{n} C_{c,exe,u\&u} \end{aligned} \tag{10}$$

The primary cost factor for cloud-based SWC is the cost of the Cloud Service Provider (CSP) $C_{c,exe,CSP}$. The cost models vary depending on the model of cloud computing and the cloud service provider. Microsoft Azure provides a comprehensive overview of the different options [18]. One commonly used pricing model is pay-as-you-go, where users are billed based on their actual usage of cloud resources. This flexible payment approach enables organizations to dynamically scale their infrastructure according to their needs. Internal operational costs $C_{c,exe,int}$ include maintenance, support and addressing downtime incidents. The costs associated with usage $C_{c,exe,use}$ include those for end-user operation, primarily data transmission charges for a data contract. Additionally, the cost of electricity used to operate the communication module must also be paid and are part of the usage costs. Costs associated with updates and upgrades $C_{c,exe,u\&u}$ are crucial for maintaining the functionality, security and performance of the cloud-based function over time. These costs may include fees for acquiring new software versions, migrating data, testing compatibility and deploying updates across the cloud environment.

C. TCO reduction options

The reduction of the TCO of a vehicle SWC can be implemented by the following three use cases:

1) Saving of a complete ECU in an existing E/E architecture: Assuming that an existing ECU in the vehicle does not have any design-related hardware proximity (e.g., due to installed I/Os) and only a single offloadable SWC is currently deployed on this ECU, it is conceivable that the ECU can be completely omitted and replaced entirely by cloud resources. In this case, the complete CapEx of the ECU will be saved and replaced by OpEx of the cloud. Due to the increasing integration of software components on an ECU as described in section II, this case is very unlikely.
2) Downsizing of an ECU in an existing E/E architecture: By relocating individual SWCs that are executed on an existing ECU to the cloud, the ECU can be re-dimensioned. This means, for example, that less computing power may be required for the ECU, meaning that a lower-performance model within a series can potentially be installed in the vehicle. Thus, saving potentials only arise through cost savings in smaller models of a ECU series.
3) The cloud as a new execution platform alternative for new E/E architectures: When designing a new E/E architecture that is not restricted to the vehicle but allows the cloud as a possible execution location, a TCO comparison needs to be executed for cloud-suitable functions or SWC.

It is important to note that these examples are not exhaustive, but rather provide an indication of the diverse approaches to TCO reduction in the development and management of SWC.

V. Case study

The proposed TCO model is assessed with a new function for electric city buses. The function is described in [19] and proposes a cloud-based machine learning model that is able to predict the Heating, Ventilation and Air Conditioning (HVAC) energy consumption with different temperature set values.
As this is a new function, it is assigned to the use case 3 "cloud as new execution platform for new E/E architectures" and the two execution platforms need to be compared with the TCO model.

A. Onboard SWC

1) Development: Software development for this new SWC in the vehicle can be estimated at 150,000 € while hardware development is one third of this amount.
2) Deployment: Adding the new SWC to the vehicle's on-board E/E architecture adds the need for computing resources that are currently not available. Therefore, a new ECU would be needed. The cost of a high-performance ECU with machine learning support sums up to 30 €. Additional hardware savings as proposed in subsection IV-A are not feasible with the function.
3) Execution: The main cost driver is the cost of OTA updates. The cost of executing the onboard SWC is rounded to 2 € per month for four updates in a one-year period [20].

B. Cloud-based SWC

1) Development: The software development costs of the SWC for the cloud can be set lower than for the vehicle. This can be attributed to the fact that it is not necessary to rely

on automotive-specific runtime environments, but instead common tools can be used for development on standard hardware. For this reason, $C_{c,dev,sw}$ is set to 90,000 €. Introduction, purchasing and migration cost add up to an equal amount.

2) Deployment: Deployment costs are negligible as they are converted into operating expenses.
3) Execution: The Microsoft Azure "Cloud services" offer a suitable configuration named *A0* for our function [21]. The function we have designed takes approx. 20 seconds to calculate for one call. We designed the function to be called every 300 seconds, such that we assume the *A0* configuration is able to handle 10 calls representing 10 vehicles within the defined time frame of 300 seconds. $C_{c,exe,CSP}$ result in the *A0* configuration monthly costs divided by 10 which is 1,49 € per month. Operation, usage and update/upgrade costs sum up to 1€ per month. Again, this is based on updating 4 times annually.

TABLE I. OPEX AND CAPEX FOR CLOUD AND VEHICLE EXECUTION PLATFORM

Exe Platform	C_{Dev}	C_{Depl}	C_{Exe}
Vehicle	200,000 €	30 €	2 €/month
Cloud	180,000 €	0 €	2.49 €/month

The life cycle of a city bus in Germany is 8.3 years [22]. The TCO for both execution platforms for this timeframe calculate to:

- $TCO_{v,SWC} = 200,229.20$ €
- $TCO_{C,SWC} = 180,247.92$ €

A break-even analysis for both execution platforms, both with and without development costs, shows the following periods after which the onboard SWC is again the cheaper option. In this case, the development costs were divided between a bus fleet of 1,000 respectively 5,000 city buses (see Figure 2):

- Break-even for 1,000 respectively 5,000 buses without C_{dev}: 61 months (5.1 years)
- Break-even for 1,000 buses with C_{dev}: 102 months (8.5 years)
- Break-even for 5,000 buses with C_{dev}: 69 months (5.8 years)

VI. DISCUSSION

The role of cloud computing in the automotive sector is becoming increasingly important as OEMs look for innovative ways to improve the functionality and user experience of their vehicles. In the future, many vehicle functions will be cloud-based, making it possible to process data in real time, perform updates and offer personalized services. Naturally, the issue of costs plays an important role here. The TCO model in this paper offers an approach to gain an overview of the costs to be expected when applying a cloud or an onboard SWC. The model enables a cost comparison and thus the inclusion of costs in the decision-making process as to where the SWC should be implemented. In Section IV-C, three approaches

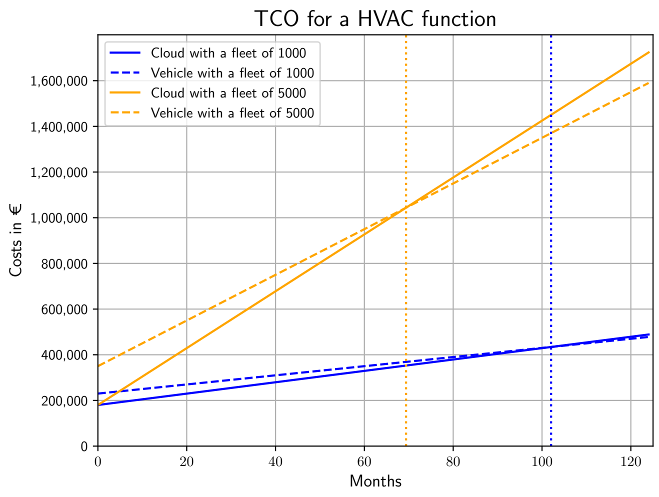

Figure 2. TCO for the HVAC function with a fleet of 1,000 (blue) and 5,000 (orange) city buses. The break-even points are marked with the horizontal dotted line.

for cost savings are presented. Saving a complete ECU is, as already mentioned, unlikely. Downsizing a current ECU by moving SWC to the cloud only has a small financial effect, in the single-digit euro range. This use case is therefore also rather unlikely. The most likely use case: the cloud as the new execution platform is in detail described in Section V. The main costs of a cloud-based SWC for the OEM are the monthly costs for the CSP, which depend on the type of service used. In addition, the OEM has the option of becoming a CSP and hosting its own cloud infrastructure. This can offer strategic advantages as the OEM has more control over its data and services, can better control security and potentially save costs, especially for long-term use. Implementing its own cloud infrastructure, allows the OEM to better meet specific performance and data protection requirements. The OEM also has the data that can be used for the development of new functionalities, for example. Within the use case (Section V) described, a fleet size of 1,000 and 5,000 vehicles is analyzed. Economies of scale must be considered for smaller and larger fleets, as they affect the cost structure. The larger the fleet size, the sooner the break-even point is reached where the aggregated cost of the cloud-based SWC becomes higher than the onboard SWC. Here, the monthly costs for deployment and execution are assumed to be constant, but as the fleet size increases, the costs for ECUs and CSPs shrink [17]. But the scaling factor of cloud costs is higher than that of onboard costs. The described case study focuses on buses, whereas a OEMs fleet of passenger cars is much larger and the economies of scale are therefore of a different order of magnitude.

While comfort functions such as HVAC systems are not considered safety-critical, network failures can still result in a negative user experience. In the scenario where functionality is lost due to network issues, the cost can be seen in terms of user frustration and potential loss of confidence in the product. To address this, manufacturers could implement fallback mechanisms, such as offering offline modes for critical features.

The increasing spread of cloud-based SWC is accompanied by a shift from CapEx to OpEx, which will have an impact on OEMs. This transition reduces upfront costs, allowing OEMs to allocate their financial resources more efficiently. Furthermore, cloud computing enables OEMs to dynamically scale their operations in response to changing demand. Cloud-based services represent also a viable strategy for end customers, who may opt for monthly subscription payments as needed. This approach provides original equipment manufacturers (OEMs) with a continuous cash flow.

VII. Conclusion and Future Work

This paper presents a TCO model that determines the total costs of a vehicle SWC. A distinction is made between two execution platforms, namely cloud and onboard. Three options are presented for reducing the TCO of a SWC. The case of a new execution platform, i.e. the relocation of an SWC to the cloud, was examined using a machine learning function in the city bus sector. Taking development costs into account, the break-even point for executing the function in the cloud is nearly the approximate lifespan of a city bus in Germany.

Although the presented TCO model offers a comprehensive framework for evaluating the total costs associated with vehicle SWCs across different execution platforms, there are several areas that require further investigation. The current model could be enhanced by incorporating additional dynamic cost factors, such as fluctuating energy prices, varying cloud service fees, and changes in hardware costs. A more dynamic model would permit real-time cost assessment and more accurate forecasting. Further research is needed to explore the scalability and adaptability of the TCO model. This includes examining how well the model performs in different organizational contexts and identifying ways to customize the model for specific business needs.

Acknowledgment

The authors would like to thank the Ministry of Science, Research and Arts of the Federal State of Baden-Württemberg for the financial support of the project INDU2 - OTRACE within the InnovationsCampus Future Mobility.

References

[1] J. P. Trovao, "Trends in automotive electronics [automotive electronics]", *IEEE Vehicular Technology Magazine*, vol. 14, no. 4, 2019. DOI: 10.1109/MVT.2019.2939757.

[2] M. Sommer, D. Baumann, T. Rösch, F. Dettinger, E. Sax, and M. Weyrich, "Process for the identification of vehicle functions for cloud offloading", in *Science and Information Conference*, Springer, 2024.

[3] T. Streichert and M. Traub, *Elektrik/Elektronik-Architekturen im Kraftfahrzeug: Modellierung und Bewertung von Echtzeitsystemen*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-25478-9.

[4] A. Bucaioni and P. Pelliccione, "Technical architectures for automotive systems", in *2020 IEEE International Conference on Software Architecture (ICSA)*, IEEE, 2020. DOI: 10.1109/ICSA47634.2020.00013.

[5] M. Traub, A. Maier, and K. L. Barbehön, "Future Automotive Architecture and the Impact of IT Trends", *IEEE Software*, vol. 34, no. 3, 2017. DOI: 10.1109/MS.2017.69.

[6] S. Frohn and F. Rees, *From signal to service: Challenges for the development of autosar adaptive applications*, Vector Informatik GmbH, Ed., Mar. 2018.

[7] E. Sax *et al.*, "Analyse der Aktivitäten und Entwicklungsfortschritte im Bereich der Fahrzeugelektronik mit Fokus auf fahrzeugeigene Betriebssysteme", *Themenpapier Cluster Elektromobilität Süd-West*, 2020.

[8] F. Milani and C. Beidl, "Cloud-based vehicle functions: Motivation, use-cases and classification", in *2018 IEEE Vehicular Networking Conference (VNC)*, IEEE, 2018.

[9] International Organization for Standardization, *Iso/ iec/ ieee 24765:2017: Systems and software engineering — vocabulary*, 2017.

[10] P. M. Mell and T. Grance, *The NIST definition of cloud computing*, Gaithersburg, MD, 2011. DOI: 10.6028/NIST.SP.800-145.

[11] L. M. Ellram, "Total cost of ownership: An analysis approach for purchasing", *International Journal of Physical Distribution & Logistics Management*, vol. 25, no. 8, 1995.

[12] P. Rosati, F. Fowley, C. Pahl, D. Taibi, and T. Lynn, "Right scaling for right pricing: A case study on total cost of ownership measurement for cloud migration", in *Cloud Computing and Services Science: 8th International Conference, CLOSER 2018, Funchal, Madeira, Portugal, March 19-21, 2018, Revised Selected Papers 8*, Springer, 2019.

[13] B. Martens, M. Walterbusch, and F. Teuteberg, "Costing of cloud computing services: A total cost of ownership approach", in *2012 45th Hawaii International Conference on System Sciences*, IEEE, 2012. DOI: 10.1109/HICSS.2012.186.

[14] M. M. Kashef and J. Altmann, "A cost model for hybrid clouds", in *Economics of Grids, Clouds, Systems, and Services*, ser. Lecture Notes in Computer Science, K. Vanmechelen, J. Altmann, and O. F. Rana, Eds., vol. 7150, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-28675-9{\textunderscore}4.

[15] M. Walterbusch, B. Martens, and F. Teuteberg, "Evaluating cloud computing services from a total cost of ownership perspective", *Management Research Review*, vol. 36, no. 6, 2013. DOI: 10.1108/01409171311325769.

[16] S. Heinrich, N. Kreft, T. Schuster, and R. Volz, "A total cost of ownership model for cloud computing infrastructure", in *Proceedings of the 56th Annual Hawaii International Conference on System Sciences*, T. X. Bui, Ed., Honolulu, HI: Department of IT Management, Shidler College of Business, University of Hawaii, 2023.

[17] J. Schäuffele, *Automotive Software Engineering: Grundlagen Prozesse Methoden und Werkzeuge effizient einsetzen*, 6. Wiesbaden: Springer Vieweg, 2016. DOI: 10.1007/978-3-658-11815-0.

[18] Microsoft, *Pricing calculator*, https://azure.microsoft.com/en-us/pricing/calculator/, Retrieved: 21.10.2024, 2024.

[19] D. Baumann, M. Sommer, F. Dettinger, T. Rösch, M. Weyrich, and E. Sax, "Connected vehicle: Ontology, taxonomy and use cases", in *2024 IEEE International Systems Conference (SysCon)*, IEEE, 2024.

[20] S. Abuelsamid, *Automotive over-the-air updates: A cost consideration guide*, Guidehouse Insights, Ed., 2021.

[21] Microsoft, *Cloud services – preise*, https://azure.microsoft.com/de-de/pricing/details/cloud-services/#pricing, Retrieved: 21.10.2024, 2024.

[22] K. Bundesamt, *Fahrzeugzulassungen*, https://www.kba.de/SharedDocs/Downloads/DE/Statistik/Fahrzeuge/FZ15/fz15_2022.pdf?__blob=publicationFile&v=5, Retrieved: 21.10.2024, 2022.

A Quantitative and Qualitative Comparison of Machine Learning Inference Frameworks

Egi Brako
Department of Computer Science
Georg-August-Universität Göttingen
Göttingen, Germany
e-mail: egi.brako@gmail.com

Jonathan Decker
Department of Computer Science
Georg-August-Universität Göttingen
Göttingen, Germany

Julian Kunkel
Department of Computer Science
Georg-August-Universität Göttingen
Göttingen, Germany

***Abstract*—As Artificial Intelligence (AI) continues to advance and impact diverse fields, ensuring universal access to its abilities becomes increasingly crucial. To make AI models accessible to users, they must be deployed to process inference requests. We conducted qualitative and quantitative analyses of popular open-source serving frameworks by evaluating their performance on three Machine Learning tasks. This research aims to shed more light on the frameworks' respective strengths and weaknesses, consequently addressing the challenges posed by the process of selecting a method of serving the models. The qualitative comparison is carried out by taking into account the subjective characteristics of each framework and scoring them on a number scale. We then use Locust to run load-tests on these frameworks, analyse their quantitative results, and compare them with each other. Our results find that PyTorch TorchServe is the overall best-performing framework, consistently surpassing the other two in our performance test. We find that some platforms have issues handling more complex models, showing incapabilities for handling specific Machine Learning tasks. Our findings show significant differences among the frameworks, contributing valuable insights for developers and researchers in selecting the most suitable framework serving Machine Learning models.**

Keywords-artificial intelligence; inference engines; machine learning;

I. Introduction

Machine Learning (ML) has emerged as a pivotal technology across various domains, revolutionizing industries such as transportation, healthcare, and finance. With the growing reliance on ML models for critical decision-making and everyday applications, the need to effectively serve these models to a wider audience has become increasingly important. Model serving refers to the deployment, management, and maintenance of ML models in production environments, ensuring their availability, responsiveness, and accuracy in delivering predictions or results in real-time.

Serving frameworks, also known as inference frameworks, play a crucial role in this process by facilitating the deployment of ML models at scale. These frameworks manage multiple requests, optimize computational resources, and enable seamless model updates. Despite their significance, selecting the right serving framework remains a challenge due to varying requirements and the distinct features each framework offers.

This paper aims to provide a comprehensive comparison of three popular ML serving frameworks: TorchServe[1], TensorFlow Serving[2], and Triton Inference Server[3]. By evaluating both performance metrics, such as latency and throughput, and usability factors, including user-friendliness and documentation quality, this research seeks to identify the most suitable framework for different ML tasks.

The remainder of the paper is organized as follows: Section II reviews related works, highlighting previous studies, advancements, as well as gaps in the field. Section III details the methodology, including the experimental setup and evaluation metrics. The results of the experiments are then presented in Section IV followed by an evaluation and discussion of the findings in Section V. Finally, Section VI concludes the paper with a summary of the research and suggestions for future work.

II. Related work

As the field of ML grows, the focus extends beyond individual models and their training to include the efficient deployment and serving of these models to users. Previous studies, such as *MLPerf Inference Benchmark* [4] and its succeeding research *The Vision Behind MLPerf: Understanding AI Inference Performance* [5], have made significant strides in establishing benchmarks for ML inference, focusing on key performance metrics and trade-offs between accuracy and performance. These works laid the groundwork for evaluating ML models across various applications, though they were limited by the scope of models and scenarios considered. For instance, models like Bidirectional Encoder Representations from Transformers (BERT) [6] and transformers [7] were not included, leaving room for further exploration of these applications.

In the area of custom serving systems, works like *Clipper: A {Low-Latency} online prediction serving system* [8] provided foundational insights into real-time ML predictions, with a focus on modularity and performance optimization techniques. However, Clipper's evaluation was limited to specific benchmarks, and it is no longer maintained, making its findings somewhat outdated.

Edge computing research, as reviewed in *Deep Learning With Edge Computing: A Review* [9] or *Edge Computing: Vision and Challenges* [10] has also been extensive, focusing on bringing Deep Learning computations closer to end devices for tasks like computer vision and Natural Language Processing (NLP). While valuable, this body of work is more concerned with inference on edge devices rather than general-purpose serving frameworks.

Our research fills the gap by providing a comprehensive, up-to-date comparison of widely-used ML serving frameworks, offering critical insights for selecting the most suitable platform for diverse ML applications.

III. Methodology

This research aims to evaluate the performance and usability of different machine learning inference frameworks. The challenge here lays in merging these aspects to form a comprehensive research question. The common part that stands out is this broad concept of "usefulness". This has led us to examine both aspects separately, and to form individual evaluations for them. Both performance metrics and the practical, user-oriented aspects contribute to determining how useful these frameworks are, under various conditions, for different users.

A. Performance

For the performance evaluation, we need to define formal methods and metrics to find out the best-performing framework. For this, we chose two different methods of load-testing (scenarios) for each serving framework: multi-stream load test, and single-stream throughput test. The former works by regularly querying the serving platform, and firing the next request as soon as the previous one completes. This will keep the platform under constant heavy load, simulating a worst-case scenario helps ensure the system is reliable, even under heavily demanding circumstances. On the other hand, the single-stream test measures how well the platform handles a continuous flow of requests, one at a time, as quickly as possible. It helps determine the maximum rate at which the platform can process inference requests without any delays or slowdowns. In both of these scenarios, three different metrics will be measured, namely: Throughput (in Requests per second), Latency (in milliseconds), and Failures (in failures per second). These measurements will help us understand how well the systems work in different situations.

B. Usability

For the usability evaluation, we must define formal metrics and strictly evaluate the serving frameworks, as it is difficult to conduct without set criteria. Our usability analysis focuses on how effortlessly a user can set up and serve the chosen models, along with the complexity of the serving process. Given that all of the frameworks under review are considerably sized projects, it is reasonable to expect a high level of support for users, both in the documentation and through the community. It is difficult to judge qualitative characteristics in a specific way, therefore we have written criteria, which will serve as our reference parameters throughout our analysis.

The reference parameters throughout our analysis will be five loose criteria that we have defined, which we will use to judge the frameworks' qualitative characteristics. Namely, these criteria are User-Friendliness, Documentation Quality, Project Features, Community Support, and Maintainance and Update Frequency. They will all receive a score based on our experience with the framework ranging from 1 to 5. A higher score indicates better usability, with a score of 5 given to a framework that is intuitive to set up and deploy, with comprehensive and understandable documentation, active community support, helpful features for model customization, and frequent updates. Conversely, a score of 1 would indicate significant problems in usability, such as convoluted setup processes, incompatibility with specific models, outdated documentation, no community engagement, or infrequent updates. Through this, a systematic comparison can be made based on the observations and findings made throughout this research. By conducting such a detailed analysis, we can present a comparison that highlights the strengths and weaknesses of each framework.

C. Workloads

In order to test these frameworks, we had to choose models and tasks that represent diverse ML applications and tasks to ensure as comprehensive a comparison as possible. When choosing the ML tasks, we focused on ones that are relevant to the field, and are different in nature from each other. We decided on three important ones, namely: Image Classification, Automatic Speech Recognition, and Text Summarization.

Image Classification is the task of analyzing a picture and automatically identifying and labeling the objects or subjects it contains. There are many well-known benchmarks and datasets in this field that allow us to compare performance, which provide widely accepted metrics, making it easier for comparison. For this task, we chose the ResNet50 [11] model. It is a variant of a Convolutional Neural Network (CNN), explicitly designed for image classification tasks. Due to its popularity in both research and industry, ResNet50 serves as a dependable standard for assessing the effectiveness of new algorithms or techniques in image classification.

Automatic Speech Recognition is the task of transforming a spoken language into written text. This involves analyzing the sound waves in a voice file and transcribing them to text. For this task we have chosen the Wav2Vec2 [12] model, specifically Wav2Vec2-base-960h, which is a model pre-trained and fine-tuned on 16kHz sampled speech audio taken from the LibriSpeech [13] dataset. Wav2Vec2 is a good choice due to its self-supervised learning being able to directly find useful representations from raw audio inputs and provides a fair ground to investigate different ML inference frameworks' capabilities without the influence of feature extraction techniques. Comparatively, other models might require extensive preprocessing or feature extraction techniques, which adds complexity and may introduce biases.

Text summarization is a task in NLP that involves shortening a text in a way that captures the main points or themes of the original text. It is a sequence-to-sequence task, which can shed light on how different frameworks handle and perform with high-dimensional, textual data. The chosen BART [14] model is designed for several NLP tasks, such as text generation, translation, and summarization. It functions by first corrupting the text and then learning to reconstruct it, working bi-directionally. The "-large-CNN" variation is specifically trained on article and summary pairs from the CNN and Daily Mail dataset. This model tackles the challenges of language understanding, context preservation, and summarization in the broad field of NLP.

It is important to mention that all the model implementations are taken as-is, directly from the respective model zoos, such as the Torchvision and the Keras packages. In some cases, the pre-trained model weights were not available in the model zoos, in which case we took their official implementations in Huggingface. Both PyTorch and TensorFlow have different formats for the pre-trained model weights, meaning they had to be saved separately. Usefully, NVIDIA Triton Inference Server serves models from both these frameworks and accepts any format of model weights that are accepted by the individual frameworks.

D. Benchmarking Environment

After downloading and saving the model files in their respective formats, we have to create our load-testing solution. Locust was our platform of choice for multiple reasons. It offers distributed load generation, meaning that all the events and virtual users can scale to multiple processes, and even multiple processors. This is very important in our case since we do not want to overload the model serving platform by running hundreds of virtual users in the same processor, as this might lead to inconsistent data. We can take advantage of this since we have plenty of resources due to our use of High Performance Computing (HPC). All experiments were conducted on a NVIDIA Quadro RTX 5000.

After choosing our tasks, models, and load-testing method, we have to consider putting all of these together in systematic, reproducible experiments. Due to the fact that we are running all the tests in an HPC environment, we chose Singularity to containerize our frameworks. In every experiment, one container serves the model, and another runs our load-tests. We have separate containers for each framework, where we bind the respective models when it is time to test them. All of the containers have their own configuration, which are easily reproducible due to the Singularity definition files. Every experiment started with both containers being deployed using SLURM batch files. To gather data from the experiments, we ran each test 5 times, saved the results in a csv file, and took the average of all runs. We chose this to increase the reliability and reproducibility of the results, minimizing the impact of outliers and background noise. After this step was complete, we raised the number of virtual users created by our load-testing container, and repeated the previous steps, until the experiment for a single model was complete.

Our chosen method not only increases reproducibility of our results, but also provides more data for statistical analysis, allowing us to understand the range and standard deviation of the results, strengthening the conclusions drawn from the study.

IV. Results

All in all, the range of the recorded results was small, with the only exception of this being the Wav2Vec2 model running on the Triton (TensorFlow) framework (as shown in Figure 1). This small range shows that our experimental results were consistent, suggesting that variations in the results can be attributed to actual performance differences rather than methodological inconsistencies or random errors.

In Figure 1, we can see the prediction latency for all the models when testing the framework performance with only one virtual user (our single-stream test). We have chosen to show only the graphics of the tests with 1 user, since the *prediction* latency is more or less similar for a specific model and framework combination, no matter the user load. The reason why the *value* of the latency increases when raising the user count is because all submitted requests must wait in the queue until all others before it have finished. Since the throughput stays constant, raising the number of incoming requests (user count) will raise the amount of time that a request waits in the queue. This phenomenon can easily be verified by taking a look at the internal logs of the serving frameworks, which show the precise inference time.

A. ResNet50

For ResNet50, PyTorch consistently outperformed other frameworks in terms of throughput across all user counts. That being said, Figure 2 shows that Triton's (PyTorch) performance is trailing not too far behind, whereas TensorFlow Serving is in this case definitively slower.

When considering all of the frameworks, the reported throughputs for the ResNet50 model are generally good. Apart from TensorFlow Serving, the fact that the other frameworks' throughputs are (consistently) around the 100 RPS (Requests per Second) mark is really good. It shows that they are able to handle a very large number of requests at the same time without any slowing down.

Similar to the throughput, PyTorch exhibited a higher comparative performance when considering latency, followed

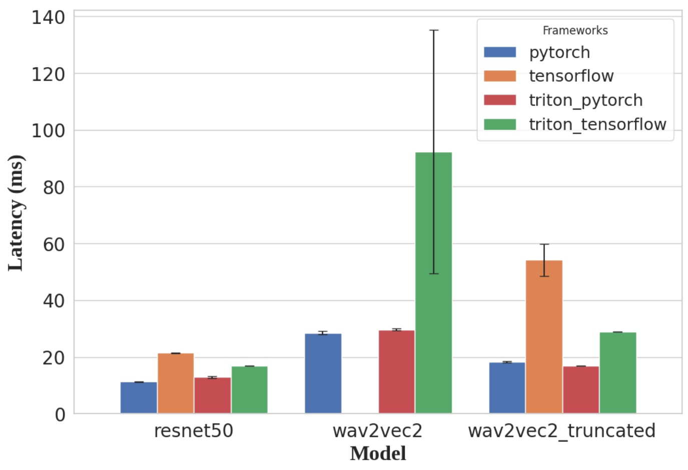

Figure 1. Latency results for 1 virtual user (in milliseconds)

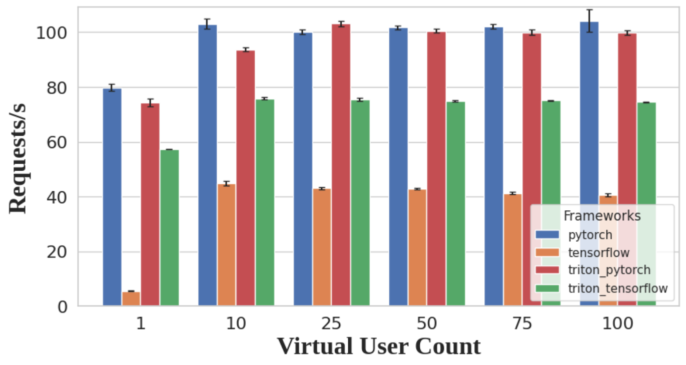

Figure 2. Throughput of the ResNet50 model (in requests per second)

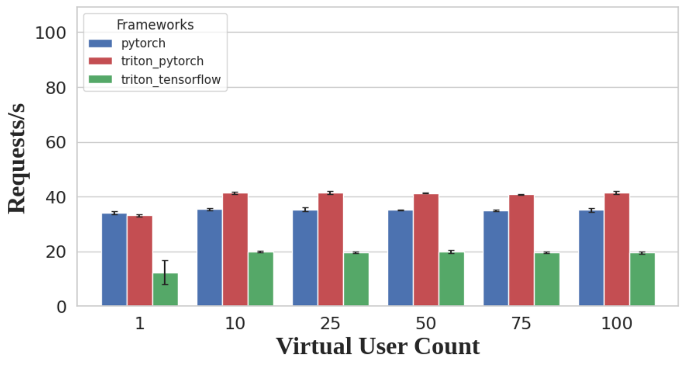

Figure 3. Throughput of the Wav2vec2 model (in requests per second)

closely once again by Triton (PyTorch). TensorFlow showed the highest latency of all the frameworks. The same relationship between the frameworks can be seen in the test with 100 users, although the numbers change slightly in user counts 10-75.

These latency results, although they differ from each other, show that the frameworks' general performance is excellent. With Pytorch and Triton, the average processing time for any request is under one second, even when considering 100 concurrent users. TensorFlow and Triton (TensorFlow) also offer a worst-case response time of around 1.3 seconds. In a realistic scenario, even if the system were under the heaviest load, this would be a very short time to wait for a response. This can be attributed to the speed and efficiency of the frameworks, but also to the small size of the model, which will become evident later, when we examine the other models.

B. Wav2Vec2.0

Earlier on, we imposed the usability constraint that the model should be taken as-is, from official resources of the frameworks. Due to this, it was impossible for us to get the base Wav2Vec2 model running in TensorFlow. Therefore, the performance analysis for this model will not feature the TensorFlow Serving framework.

When it comes to throughput, we can tell from Figure 3 that Triton and PyTorch displayed comparable performance at lower user counts, but Triton excelled as the load increased. In the test with 10 and more virtual users, all three frameworks' throughputs stay constant, implying that this is caused by the models themselves, and not by the user count.

Although we are comparing the frameworks to each other, we should also consider the absolute values of the throughputs. Triton (TensorFlow) being able to concurrently process 20 requests at all times is no small feat. This is much more significant when considering the performance of Triton (PyTorch), which processes almost double the requests. Overall, we can say that the throughput of the frameworks when observing them individually is excellent.

An interesting observation is that Triton's (TensorFlow) prediction latency is (in comparison) quite high, starting from the test with only one virtual user. This gets considerably worse the more users are added to the load test, peaking at almost double that of Triton's (PyTorch) prediction latency in the test with 100 concurrent users.

This being said, in the test with one user, all frameworks show a very small response time for most requests (under 100 ms). This makes it quite applicable to the task of real-time text-to-speech transcription. However, as the concurrent users requesting the Automatic Speech Recognition (ASR) service increase, we can see the latency dramatically increasing. Although the processing time for each individual request remains relatively similar, the latency is higher due to their waiting time in the queue. This shows that under heavy user load, performance may degrade to the extent that the service becomes unusable. Although this should be taken with a grain of salt, since in real scenarios, the platform is not going to constantly be under the heaviest load.

C. Wav2Vec2.0 truncated

The truncated version of Wav2Vec2 was implemented to accommodate the limitations of the official TensorFlow implementation of the Wav2Vec2 model. This workaround involved truncating (or padding) the inputs to be able to continue our performance evaluation across all frameworks. This approach results in incomplete outputs, as the truncated inputs do not provide the full context necessary for full model predictions, resulting in sentences seemingly cutting off. Despite these obstacles, we believe the analysis offers interesting results into the performances of the different frameworks.

Figure 4 gives an overview of the throughput results for the Wav2Vec2 truncated model. This model exhibits the same behavior as was observed in the base Wav2Vec2 model. The truncated inputs, which are significantly shorter than the normal Wav2Vec2 model inputs, reinforce the evidence from the base Wav2Vec2 model that NVIDIA Triton provides much better performance. As before, we can see here the throughput plateauing in the tests with more than 10 users. This time, however, due to the input being significantly shorter, the absolute value is higher. Interestingly though, we can observe that TensorFlow performs slightly better than PyTorch and much better than Triton (TensorFlow), which is quite a difference from the behavior observed thus far.

When considering the throughput values for each framework individually, we can see that the values are 1.5 to 2 times

TABLE I. FRAMEWORKS' PERFORMANCE, BASED ON THROUGHPUT AT 100 USERS (IN RPS)

Frameworks	PyTorch Torchserve	TensorFlow Serving	Triton (PyTorch)	Triton (TensorFlow)
ResNet50	**104.10**	40.59	99.74	74.53
Wav2Vec2	35.16	-	**41.43**	24.51
Wav2Vec2 (truncated)	51.24	69.34	**83.14**	41.14
BART	**1.44**	-	-	-

better than their respective values from the original Wav2Vec2 model. Due to the fact that all the inputs were truncated in order to fit this model (meaning a large number of them were incomplete), we believe it is impossible to reach a consensus of whether these values would be useful for any real-world application.

D. Issues and Challenges

Before we evaluate the results of our experiments, we have to bring up the challenges faced throughout. As mentioned above, TensorFlow could not serve Wav2Vec2 due to constraints in TensorFlow Hub[15]. The SavedModel's serving signature requires an input tensor of size `(-1, 50000)`, meaning audio sequences must have 50,000 samples. Our attempts to define a custom input format failed, possibly due to a specific operation in the TensorFlow graph, specifically in the convolutional layer in the positional convolutional embedding `(pos_conv_embed)` of the Wav2Vec2 encoder.

After multiple failures, we decided to align all frameworks with the TensorFlow model's constraints. To match the required input size, we truncate or pad the audio, which allows cross-framework performance evaluation but introduces inaccuracies. This (unusual) truncation process itself should ideally not be included in the performance metrics, as it is part of preprocessing rather than model inference. However, since we are looking for an overall comparison of the entire model inference pipeline and all the inputs are being truncated in the same method, we are considering this step part of the preprocessing.

We faced additional challenges with serving the BART model, which can be generalized to generative models as well. Triton requires models to be converted to TorchScript, using PyTorch's JIT compiler to optimize and interpret them at runtime. However, the BART model's `.generate()` function has dynamic operations that are difficult to trace due to variable-length loops and control flows not handled well by `torch.jit.trace`. This method of tracing essentially captures the operations executed over a single forward pass to create a static graph, hence failing to correctly trace operations with dynamic control flow.[16]

Serving the BART model in TensorFlow faced similar challenges due to the `.generate()` method's loops and conditionals that do not translate well to a static graph format. Issues with TensorFlow Serving arose due to dynamic tensors created by the method, which conflicts with the XLA[17]. These issues with BART highlight a serious limitation: the static graph requirements of these frameworks make it hard to handle models with complex control flows, such as generative models. We could only serve BART with PyTorch, as TorchServe does not require JIT compilation and allows custom model handlers to call the `.generate()` method during handling.

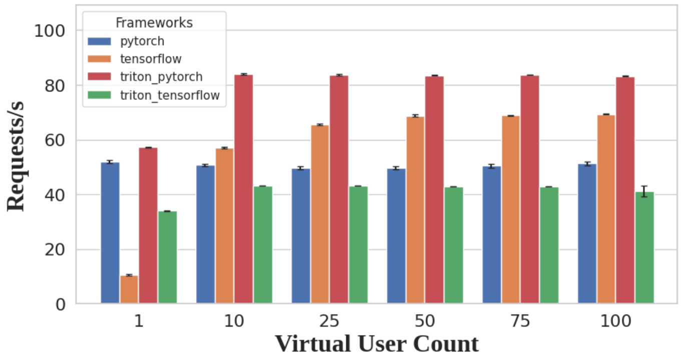

Figure 4. Throughput of the Wav2Vec2 Truncated model (in requests per second)

V. DISCUSSION | EVALUTION

A. Evaluation

Judging from the individual model graphs as well as Figure 1, we can see that PyTorch is overall the best when measuring prediction latency. From the same figure we can also tell that the range of the latency results for all (but one) frameworks was quite stable, thus proving that the results are accurate.

Regarding the throughput, the results were consistent across different user counts, with PyTorch and Triton (PyTorch) competing closely for first place. The throughput performance in Table I also points to the fact that the performance of TensorFlow and Triton (TensorFlow) is consistently the worst, in most cases performing around 0.5x than the best framework. Nevertheless, Triton Inference Server showed a good capability to increase the prediction speed for TensorFlow models, bringing it closer to that of PyTorch.

Since PyTorch consistently shows the lowest latency in the chosen models, it is ideal for real-time applications. Whereas PyTorch excels in low latency, Triton (PyTorch) stands out for high throughput across all models, only trailing behind the former framework.

On the usability side, scores ranging from 1-5 were assigned to each serving framework based on our personal experience and observations. The scores align with the usability criteria detailed in Section III.

The scores shown in Table II are crucial for adressing the question of which is the most usable framework. PyTorch emerges as the most usable framework, scoring nearly perfect across all criteria. Its relatively easy setup, comprehensive documentation, and features set it apart from the other frameworks.

B. Discussion

Our results suggest that, although there have been quite a few differences from the Biano-AI research [18], the best-performing framework is still PyTorch. Nevertheless, the failure rate in our research was completely different from the preceding one. None of the tests we conducted showed any failed requests across all frameworks and models tested, marking a significant difference from the previous study.

The individual performance of all frameworks has also improved considerably with TensorFlow Serving improving the most. This suggests that ongoing updates and community contributions are improving its capabilities, even though it still lags behind PyTorch in some aspects.

Based on the results in Table 1, it is evident that the performance of the ML inference frameworks varies significantly, with each framework exhibiting strengths and weaknesses in different areas.

The lower latency PyTorch consistently demonstrates makes it a highly suitable choice for applications where real-time predictions are crucial, such as in ASR tasks. Additionally, PyTorch exhibits high throughput, meaning it can handle a large number of requests per second without significant performance degradation. This makes PyTorch ideal for high-demand applications where both low latency and high throughput are required.

TensorFlow, on the other hand, generally shows higher latency compared to PyTorch and Triton (PyTorch). This higher latency might be a limiting factor for applications that require immediate responses. However, TensorFlow's throughput is competitive in the ASR task, especially in scenarios with high user counts. This indicates that TensorFlow can still be effective in applications where throughput is prioritized over latency, making it a viable option for certain types of high-demand environments.

Triton's throughput, particularly with PyTorch models, is the highest among the frameworks evaluated. This high throughput makes Triton highly suitable for applications that need to handle very high demand. Even when using TensorFlow models, Triton shows improved latency and throughput compared to native TensorFlow, making it a better choice for TensorFlow-based applications that require higher performance.

To apply the results, we must also understand the metrics. Our analysis pertains more to the relationship with the results, rather than the results proper. Let us consider the use case where we want to offer AI-as-a-Service. The important aspect would be to offer the users requiring this service a response from the model as soon as possible. When considering a real scenario like this, the serving framework might not be under load constantly, which means one of the main goals would be to offer as low of a latency as possible. This is why when we look at the results, we consider the latency concerning the test with a single virtual user. This is also why we would recommend the choice of either PyTorch or Triton (PyTorch), instead of the (much) slower TensorFlow Serving. However, for applications where TensorFlow's ecosystem and tools are needed, its performance may still be acceptable, especially given the significant community support and documentation available.

Alternatively, in a situation where usability is paramount for users with limited technical expertise, the choice of serving framework extends beyond performance metrics alone. Ease of setup, deployment, and maintenance are critical factors influencing usability. We identify PyTorch as the best option here, scoring the highest in our usability scores (see Table II). Its comprehensive documentation and community support ensure that users can deploy models with minimal issues and easily troubleshoot problems that arise. It should be mentioned that Triton also performs well, particularly in serving models from other frameworks and offering many features for optimization.

VI. Conclusion and Future Work

This work has provided a quantitative and qualitative comparison of various ML inference frameworks. The research question aimed to identify the most suitable framework for different use cases based on performance and usability metrics.

The methodology involved a carefully constructed approach to ensure the reliability of our findings. We selected representative models for three distinct tasks. Each model was deployed and tested on the respective frameworks under controlled conditions. Our experiments included two different types of load-tests: single-stream and multi-stream. Both performance and usability were assessed based on clear, concise criteria that we constructed.

This study's contribution lies in its in-depth analysis of the ML serving frameworks, providing valuable insights for different use cases and applications. By evaluating both

TABLE II. Usability scores (1-5). Higher scores indicate better usability.

Frameworks	PyTorch Torchserve	TensorFlow Serving	NVIDIA Triton
User-Friendliness	5	3	4
Documentation Quality	5	3	5
Project Features	5	4	5
Community support	4	5	4
Maintenance and Update Frequency	4	5	5

performance and usability, this research shows that the choice of serving framework is as critical as the selection of the model for ML tasks, proving that the serving framework can significantly impact the overall effectiveness and efficiency of the deployed model.

Our study was limited to the default configurations of the frameworks and models, therefore future work should include testing the ML models without any constraints, and exploring which frameworks would be the most effective at running the different models. Other than that, expanding the scope of future work to investigate performance across more diverse use cases, or to include novel frameworks, such as in edge computing, could provide valuable insights into the field of ML.

In conclusion, this paper has provided a thorough comparison of TensorFlow Serving, PyTorch TorchServe, and NVIDIA Triton Inference Server, showcasing their strengths and weaknesses in different scenarios. The insights gained from this research can guide users to select the most suitable framework based on particular requirements.

References

[1] PyTorch, *Github - pytorch/serve*, https://github.com/PyTorch/serve, Accessed: 2024.11.15.

[2] TensorFlow, *Github - tensorflow/serving*, https://github.com/tensorflow/serving/, Accessed: 2024.11.15.

[3] Triton Inference Server, *Github - triton-inference-server/server*, https://github.com/triton-inference-server/server/, Accessed: 2024.11.15.

[4] V. J. Reddi *et al.*, “Mlperf inference benchmark”, *Computing Research Repository (CoRR)*, vol. abs/1911.02549, 2019. arXiv: 1911.02549.

[5] V. J. Reddi *et al.*, “The vision behind mlperf: Understanding ai inference performance”, *IEEE Micro*, vol. 41, no. 3, pp. 10–18, 2021. DOI: 10.1109/MM.2021.3066343.

[6] J. Devlin, M. Chang, K. Lee, and K. Toutanova, “BERT: pre-training of deep bidirectional transformers for language understanding”, *CoRR*, vol. abs/1810.04805, 2018. arXiv: 1810.04805.

[7] A. Vaswani *et al.*, “Attention is all you need”, *CoRR*, vol. abs/1706.03762, 2017. arXiv: 1706.03762.

[8] D. Crankshaw *et al.*, “Clipper: A Low-Latency online prediction serving system”, in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, p. 615.

[9] J. Chen and X. Ran, “Deep learning with edge computing: A review”, *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019. DOI: 10.1109/JPROC.2019.2921977.

[10] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge computing: Vision and challenges”, *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–638, 2016. DOI: 10.1109/JIOT.2016.2579198.

[11] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition”, *CoRR*, vol. abs/1512.03385, 2015. arXiv: 1512.03385.

[12] A. Baevski, H. Zhou, A. Mohamed, and M. Auli, “Wav2vec 2.0: A framework for self-supervised learning of speech representations”, *CoRR*, vol. abs/2006.11477, 2020. arXiv: 2006.11477.

[13] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, “Librispeech: An asr corpus based on public domain audio books”, in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015, pp. 5206–5210. DOI: 10.1109/ICASSP.2015.7178964.

[14] M. Lewis *et al.*, “BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension”, *CoRR*, vol. abs/1910.13461, 2019. arXiv: 1910.13461.

[15] Kaggle, *Wav2vec2 model on tensorflow2*, https://www.kaggle.com/models/kaggle/wav2vec2/tensorFlow2/960h/1?tfhub-redirect=true, Accessed: 2024.11.15.

[16] PyTorch, *Torch.jit.trace – pytorch 2.4 documentation*, https://pytorch.org/docs/stable/generated/torch.jit.trace.html, Accessed: 2024.11.15.

[17] OpenXLA Project, *Openxla project*, https://openxla.org/xla, Accessed: 2024.11.15.

[18] Biano AI, *Quantitative comparison of serving platforms for neural networks*, https://biano-ai.github.io/research/2021/08/16/quantitative-comparison-of-serving-platforms-for-neural-networks.html, Accessed: 2024.11.15.

Scalable Software Distribution for HPC-Systems Using MPI-Based File Systems in User Space

Jakob Dieterle
GWDG
Göttingen, Germany
e-mail: `jakob.dieterle@gwdg.de`

Hendrik Nolte
GWDG
Göttingen, Germany
e-mail: `hendrik.nolte@gwdg.de`

Julian Kunkel
Department of Computer Science
Georg-August-Universität Göttingen
Göttingen, Germany
e-mail: `julian.kunkel@gwdg.de`

***Abstract*—Even on state-of-the-art high-performance computing systems, data-intensive tasks that are running on a lot of nodes can encounter waiting times when accessing large files. This can be caused by a bottleneck in the network bandwidth. To this end, this paper aims to develop a filesystem that utilizes inter-node communication to more efficiently distribute large files between nodes and reduce the bandwidth usage on the network. The presented file system was implemented using the Filesystem in Userspace interface and the Message Passing Interface. During evaluation, it showed promising performance compared to a slower native file system but did not reach the performance level of an optimized file system. A refined version was able to outperform the slower native file system with 16 nodes by 8%, achieving a bandwidth-reduction to latency-increase ratio of approximately 22.**

Keywords-data distribution; optimized reading; one-sided communication.

I. INTRODUCTION

With the steady increase of demand for computing power for use cases both in research and industry and the slowdown of Moore's law [1], high-performance computing (HPC) systems have to increase their number of nodes to keep up. This growing number of nodes imposes significant challenges on the I/O performance of large-scale HPC systems [2]. Distributing data, container files, or software packages to hundreds or thousands of nodes for a single job can lead to long waiting times before any processing can even begin [3]. The bottleneck in such situations is the bandwidth of the storage nodes hosting the relevant files.

The comparable low available bandwidth originates from different sources. For instance, large HPC systems are increasingly built by partitioning the overarching system into different compute islands [4]. A high-speed interconnect connects all nodes within each island, while the connections between different compute islands have a comparatively low bandwidth. Since the overarching software stack does not necessarily change drastically between the different islands, it is preferable from an administrative perspective to centrally manage and export a global software stack to the individual islands. Therefore, accessing software typically requires going through an interconnect with a limited bandwidth.

To tackle this issue, we want to develop a mechanism that reduces the bandwidth usage on the interconnect to the storage nodes by utilizing inter-node communication. Therefore, access through the remote software stack is only done once, and software distribution to potentially hundreds of nodes is done via the local, high-speed interconnect. The challenge is to find an efficient way to distribute the data between the nodes while reading the file's content only once. The best way to implement such a mechanism in a user-friendly way is to include it in a custom file system. Using the Filesystem in Userspace (FUSE) interface will allow us to develop and run the file system in user space, which is necessary with standard user privileges. We will implement the file system using the Message Passing Interface (MPI) to facilitate inter-node communication since it is the most commonly used interface for message passing. Our file system is supposed to run on the computing nodes, handling the access of files provided by an existing distributed file system. Popular examples of distributed file systems are *BeeGFS*, *Lustre*, *GPFS*.

The main contributions of this work are:

- Presenting multiple designs for a file system to reduce overall bandwidth usage to the storage nodes
- Implementation of two design approaches using different communication paradigms
- Benchmarking the implemented file systems with various configurations

The remainder of the paper is organized as follows: Section II presents the used technologies and related work. In Section III, we outline the design of the file system. Section IV is about evaluating the implemented file system, including methodology, results, and discussion. Finally, Section V contains the conclusion, and we discuss future work.

II. BACKGROUND AND RELATED WORK

To make file systems available in user space, the FUSE kernel module was incorporated into the Linux kernel with version 2.6.14 [5], which consists of the kernel module and the libfuse userspace library. By linking the libfuse library to a program, a non-privileged user can mount their own file system by writing their own open/read/write, etc. methods. This allows the implementation of custom file systems that do not necessarily require a dedicated storage device but can instead use forward storage requests to an underlying file system. For example, Fuse-archive by Google [6] allows the user to mount different archive file types (.tar, .zip, etc.) and access it like a regular directory while decompressing the data on the fly.

Rajgarhia and Gehan evaluated the performance of FUSE using the Java bindings as an example [7]. They found that for block sequential output, FUSE adds a lot of overhead when dealing with small files and a lot of metadata but becomes quite efficient with larger files. When running the PostMark benchmark, FUSE added less than 10% compared to native ext3.

Vangoor et al. also analyzed the performance of FUSE and its kernel module design in greater detail [8]. According to Vangoor et al., FUSE can perform with only 5% performance loss in ideal circumstances, but specific workloads can result in 83% performance loss. Additionally, a 31% increase in CPU utilization was measured.

The most commonly used standard for passing messages between nodes is the *Message Passing Interface* (MPI). MPI provides different concepts for communication, such as point-to-point communication (send, receive), collective communication (broadcast, gather, allgather), and one-sided communication (get, put).

The performance of MPI can vary depending on the environment and implementation that is being used. Hjelm analyzed the performance of one-sided communication in OpenMPI [9]. The paper provides an overview of OpenMPI's RMA implementation and evaluates its performance by benchmarking the `Put`, `Get`, and `MPI_Fetch_and_op` methods for latency and bandwidth. The benchmarks showed constant latency for `Put` and `Get` for messages of up to 2^{10} bytes and a drastic latency increase for messages larger than 2^{15} bytes. Analog to the latency results, the bandwidth performance plateaus with message sizes larger than 2^{15} bytes.

There are also alternatives to MPI for message passing and I/O management. One is Adios2, presented by Godoy et al. [10]. Adios2 is designed to be an adaptable framework to manage I/O on various scales, from laptops to supercomputers. Adios2 provides multiple APIs with its MPI-based low-level API designed for HPC applications. It realizes both parallel file I/O and parallel intra/interprocess data staging. Adios2 adopts the Open Systems Interconnection (OSI) standard and is designed for high modularity.

With the increasing complexity of HPC applications, the complexity and size of software packages used for these tasks also increase. Zakaria et al. showed that package dependencies in HPC have increased dramatically in recent years [11]. Different well-known software deployment models are discussed in this paper, including store models like spack, which is used on Sofja. The authors also present their solution, Shrinkwarp, which reduces loading times for highly dynamic applications. The paper focuses more on software distribution models and package management than improving loading times by increasing I/O efficiency.

Creating file systems in user space to improve I/O performance is not new. There are already several other file systems with similar goals. For example, FusionFS [12]. FusionFS is a file system in user space that supports intensive metadata operation by storing metadata for remote files locally and is optimized for file writes.

The concept of disaggregation in HPC systems aims to decouple resources like memory and processing power by allowing direct memory access over network interfaces. Peng et al. conducted a study on memory utilization, showing that 90% of all jobs utilized less than 15% of node memory and 90% of the time less than 35% of node memory is used [13].

Above, we presented existing projects that aim to improve I/O performance. However, none of those focus on enhancing performance for distributing large files from a few storage nodes to many compute nodes.

III. Design

Before designing a mechanism to distribute files with our file system, a decision has to be made on which communication paradigms MPI offers should be used. The obvious answer to that might be collective communication since, with the broadcast operation, distributing data from one node to all other nodes is very easy and efficient. However, using collective communication also imposes very difficult to overcome limitations. In case there are one or more nodes in the job that don't access a file that all the other nodes need to access, all other nodes would get stuck when trying to broadcast the data, or all nodes would have to be forced to join the broadcast, even if they don't need to access the broadcasted data. But even if all nodes want to access the same file, they would also have to access the blocks of the file in the same exact order, which is not something we can expect. Point-to-point communication can also be very efficient. However, it always needs interaction from both involved nodes, which means distributing data to a lot of nodes would have to be organized in a predetermined way. The complexity of such a mechanism would most likely scale very severely with a large number of nodes. MPI's One-Sided communication methods are known to be less efficient but allow nodes to access designated parts of the memory of the other nodes by utilizing remote memory access. With the `MPI_Get` method, we can read data from other nodes without interrupting the process on the target node. Using One-Sided communication also gives us more flexibility since we don't have to synchronize the processes between nodes, simplifying the mechanism and reducing busy waiting times. For these reasons, we decided to design a mechanism specifically using the `MPI_Get` method.

To fully use the benefits of the `MPI_Get` method during file access, we want to ensure the entire file is already available to be accessed only using direct memory access between the nodes. To that end, the file is split into N parts, with N being the number of nodes. In the `open` method of our FUSE file system, each node reads its assigned part of the file from the storage nodes. This process is predetermined for all nodes so each node can create a lookup table to know which node holds each part of the file (see Listing 1).

When a node calls the read method of our file system to access a part of the file, it will check which node or nodes are holding the needed data. If the data is already in the local buffer the data can just be returned from the buffer, otherwise, the data has to be obtained from one or more nodes using the `MPI_Get` method (see Listing 2). If data was retrieved from other nodes, it would be written to the `file_buffer`, and

```
def my_open(path):
    file_handler = open(path)

    file_buffer = MPI_Win_allocate
        (file_handler.size, char)
    meta_buffer = int[file_handler.size]

    meta_buffer = calculate_distribution
        (file_handler.size, world_size)

    offset, size = calculate_my_range
        (file_handler.size, my_rank)
    data = read(offset, size)
    file_buffer[offset:offet+size] = data

    return file_handler
```

Figure 1. Pseudo code for open method

```
def my_read(file_handler, offset, size):

    if (in_buffer(offset, size)):
        return file_buffer[offset:offset+size]

    targets = get_targets(offset, size)
    for target in targets:
        t_offset, t_size = caclulate_target_range
            (meta_buffer, offset, size)
        data = MPI_Get(t_offset, t_size, target)
        file_buffer[t_offset:t_offset+t_size] = data

    meta_buffer[offset:offset+size] = my_rank

    return file_buffer[offset:offset+size]
```

Figure 2. Pseudo code for read method

the `meta_buffer` will also be updated accordingly so that we don't have to retrieve the data multiple times.

With this design approach, the bandwidth usage to the storage nodes is minimized to a workload of just $1 \cdot filesize$, instead of $N \cdot filesize$, since each part of the file is only read once during the open method. Afterward, the nodes only communicate with each other to distribute the data. In the following sections, we will refer to this design as the One-Sided-Reading (OSR) file system. Additionally to this design approach, a simple design using a broadcast operation in the read method was implemented to compare performance between the two communication paradigms. We will refer to this implementation as Naive design approach or simple broadcast approach.

IV. Methodology

This work's main focus is read timings, which are important variables in terms of the scalability of our file systems. To evaluate the implemented file system's performance, tests will be conducted over a range of nodes and file sizes.

The tests were conducted on a smaller tier 3 HPC system. That means all nodes are shared by default and not exclusively allocated for each job. The *Sofja* System consists of 30 Nodes, with each node providing two AMD EPYC 7763 64-core processors on two sockets, totaling 128 cores per node and 3840 cores for the whole system. Each node also provides 512 GB of memory. The cluster is split up into two racks and uses HDR100-InfiniBand fabric. The nodes have access to different storage systems. The StorNext file system is used to access the 3 PiB of Home directories. For intense I/O applications, the dedicated Scratch file system can be used. The Scratch file system offers more than 500 TiB of storage space, from which more than 100 TiB are NVMe drives. The Scratch file system provides much better I/O performance and higher bandwidth than the Home directories and runs on BeeGFS. The 30 nodes that will be used for testing also provide local SSDs that offer temporary storage per job and memory-based storage on `/dev/shm`.

Six use cases will be tested by accessing files with `sha256sum`: access over the native file system, access over our One-Sided Reading file system (OSR), copying the file to local SSDs with the Naive approach before access (Bcast to local), copying the file to `/dev/shm` with the Naive approach before access (Bcast to /dev/shm), accessing the file directly with the Naive approach (Bcast no copy, possible, since `sha256sum` accesses the file sequentially like `cp`) and access over the simple FUSE Passthrough. These six use cases will also be tested with the Home directory as the original source of the file and the Scratch file system as the source. That results in twelve total test scenarios.

Each scenario will be tested on all combinations from the number of nodes and file sizes. With the number of nodes being: 1, 2, 4, 8, 16, and 24. And file sizes of 1KB, 100KB, 10MB, and 1GB. This means all ten scenarios will be tested with 24 configurations. Each configuration will be tested ten times to receive statistically robust result data. Although the cluster offers 30 nodes, we only tested with up to 24 nodes since not all nodes are available at all times.

To guarantee the resulting timings are valid, we have to ensure the accessed files can not be cached on the nodes between tests since we want to measure the time it takes to actually read the file from the storage nodes over the network. The best way to guarantee this would be to drop the page caches between runs. However, that requires sudoer rights, which we didn't have. Thus the random test data was generated for each individual test, that way the file's content changes between each test and thus cannot be accessed from caches. However, just writing the file to the desired location can cause the file to be in the page caches of the node that is generating and writing the file. To ensure this doesn't affect the measurements, we will run the benchmarker with one extra node, which only generates the file but doesn't run any tests.

All tests are organized into runs. Each run tests one combination of the number of nodes and file size for all six use cases on either the Home file system or Scratch file system. For each test run, three jobs are needed: the One-Sided Reading file system, the Naive file system, and the Python benchmarker, which executes test file generation, all tests, and writes the resulting data into a `csv` file. When testing on n nodes, the benchmarker has to be run with $n + 1$ nodes so that the first node of the benchmarker job is not included in the n nodes of the jobs for the two file systems. The benchmarker generates the test files using `/dev/urandom`. The time needed to calculate the hash sum is measured using MPI's `MPI_Wtime` method.

V. Results

The results of the described benchmarks can be seen in Figure 3, with more details listed in Table I. First, we look at the results when accessing the files over the Home directory. With small files, you can see that the native file system is the fastest, and the number of nodes doesn't affect the performance much. With the 10 MB file, the native file system starts to lose performance with a steady increase of time needed with more than four nodes. Between one node (0.1 sec.) and 24 nodes (0.23 sec.), the native file system is 2.3 times slower. The effect is more pronounced with the 1 GB file, where it is 2.91 times slower (from 7.35 sec. to 21.41 sec.). During testing with the 1GB file and 24 nodes, it achieved a throughput of 1.121 GB/s. The three different broadcast use cases also have no visible loss in performance with the 1KB file but start to lose performance beginning with the 100KB file and more than four nodes, where all three perform very similarly. The Naive design performs similarly with the 10 MB file when copying to local SSDs or to `/dev/shm`. When copying to SSDs, the procedure is 2.85 times slower with 24 nodes (0.4 sec.) compared to 1 node (0.14 sec.). When copying to `/dev/shm`, it is 3.58 times slower (from 0.12 sec. to 0.43 sec.). When accessing the file directly, the Naive design starts to be much slower, with more than eight nodes. That takes 5.69 times longer when going from 1 node (0.13 sec.) to 24 nodes (0.74 sec.). The trend is continued for the 1 GB file. From the three broadcast use cases, copying the to `/dev/shm` achieves the highest throughput on 24 nodes with the 1 GB file of 0.689 GB/s, while accessing the file directly only achieves 0.331 GB/s with the same configuration. The One-Sided Reading design is the worst file system with small files and more than two nodes. It is also the only file system that gets significantly slower with an increasing number of nodes on the 1KB file. With the 10 MB file, the scaling of this file system is already better than any of the broadcast use cases. Here, it is 2.56 times slower when going from 1 node (0.16 sec.) to 24 nodes (0.41 sec.). With increasing file size the loss in performance decreases, with the 1 GB file, it is only 1.62 times slower (from 14.28 sec. to 23.09 sec.). When accessing the 1 GB file with 24 nodes, the One-Sided Reading file system is much faster than the three broadcast use cases and only 1.68 sec. or 7.85 % slower than the native file system, achieving a throughput of 1.039 GB/s. In general, the results for the Passthrough file system are very similar to the performance of the Home file system.

When we look at the results when using the `/dbnscratch` file system, we have slightly different results for our implemented file systems. With just one node, all file systems are faster than when using `home`; with 24 nodes, all broadcast use cases and the One-Sided Reading file system are slower compared to using `home`. That is also reflected in increased multipliers when comparing the performance with one node against 24 nodes, which can be found in Table I. The Scratch file system itself however is much faster than the Home directory file system, even with a 1 GB file size it even shows a performance increase when going from 1 to 24 nodes, achieving a throughput of 16.819 GB/s. The simple Passthrough file system shows similar performance in the beginning. The better performance with small files and a few nodes can also be attributed to the deviations between nodes again. Starting with the 10 MB file, a significant difference can be observed compared to the Scratch file system for any number of nodes.

It would be interesting to know how much each factor actually affects performance. Comparing the overhead caused by MPI and FUSE would be important to identify where the file system can be improved. To this end, another test was conducted with the OSR file system and the Naive file system. With the OSR file system, the `xmp_open` method and the `xmp_read` method were timed. In the `xmp_read` method of the Naive file system, the time it takes to read the requested range of bytes was measured, as well as the time it takes to broadcast the read data. For both file systems, a counter was added to count how many times the `xmp_read` method was called on each node. Using this information we can calculate the time the operations of the two file systems take in total. For the OSR file system, we take the average of the `xmp_open`

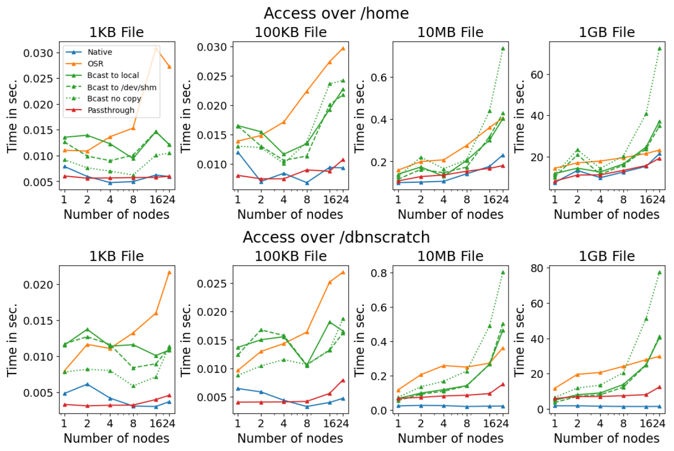

Figure 3. Time measurements on Sofja system

TABLE I. Result comparison for tests with 1 GB file

	Time with 1 node (in seconds)	Time with 24 nodes (in seconds)	Increase	Throughput with 24 nodes
/home	7.355	21.408	2.910	1.121 GB/s
OSR	14.289	23.094	1.617	1.039 GB/s
Bcast to local	11.703	37.161	3.175	0.646 GB/s
Bcast to /dev/shm	9.991	34.826	3.486	0.689 GB/s
Bcast no copy	11.124	72.505	6.518	0.331 GB/s
Passthrough	8.021	19.192	2.393	1.251 GB/s
/dbnscratch	1.772	1.427	0.806	16.819 GB/s
OSR	11.607	29.737	2.562	0.807 GB/s
Bcast to local	4.923	40.404	8.207	0.594 GB/s
Bcast to /dev/shm	3.625	41.221	11.372	0.582 GB/s
Bcast no copy	6.478	77.450	11.956	0.310 GB/s
Passthrough	5.872	12.426	2.116	1.931 GB/s

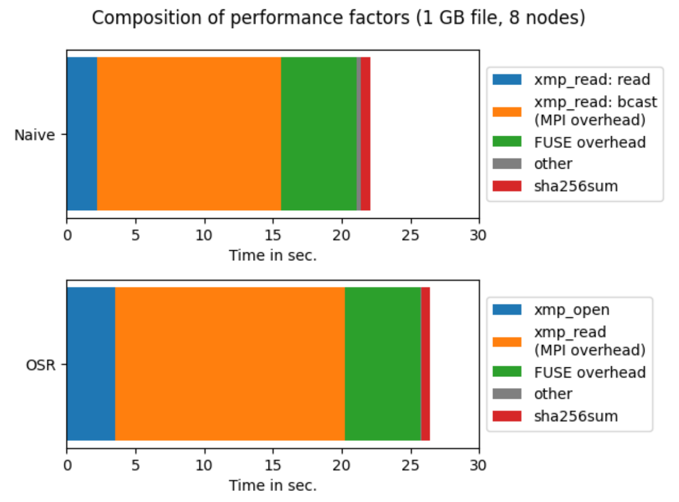

Figure 4. Composition of performance factors for the OSR and Naive file system

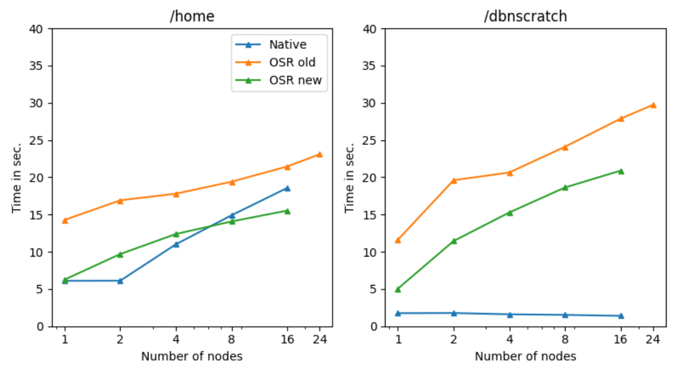

Figure 5. Timings of OSR file system without metadata buffer compared to with metadata buffer

timings over the nodes to get the value for this method. For the `xmp_read` method, we take the average over all calls of this method on all nodes and then multiply this value by the number of times the method was called on each node. The same is done for the two timings we take in the `xmp_read` method of the Naive file system. We can use the test results visualized in Figure 3 to quantify the overhead FUSE introduces. We can subtract the result for the Scratch file system from the same result for the Passthrough file system when accessing the file over the Scratch file system. This difference can be attributed to the FUSE overhead since the simple Passthrough file system does not add any features. It only passes the file system calls through the FUSE kernel module. A more detailed view of how the FUSE overhead is calculated and what it consists of can be seen in the following equation.

$$\begin{aligned} \text{FUSE-Overhead} &= time(\text{FUSE Passthrough}) - time(\text{native}) \\ &= time(\text{FUSE context switches} \\ &\quad + \text{ executing FUSE method} \\ &\quad + \text{ native FS call}) \\ &\quad - time(\text{native FS call}) \end{aligned} \tag{1}$$

Together with the result of measuring the time `sha256sum` takes to calculate the hash-sum when the file is already in the memory (0.653 seconds), we can deduct those measurements from the total time it took to calculate the hash-sum during this test, and we receive a remaining time, that should be very small and can be attributed to the fact that we combine different test results and other smaller factors that we were not able to measure. The test was conducted by accessing a 1 GB-sized file on the Scratch file system with eight nodes, as most of the system's nodes were occupied by other long-running jobs.

The total time the test took was similar to before: 21.432 seconds for the Naive file system and 26.426 seconds for the OSR file system. Both results are a little bit higher than in the previous tests; this could be caused by the job mentioned that was running concurrently. The `xmp_open` method of the OSR file system took 3.5 seconds on average over the nodes. The `xmp_read` calls took around 0.002 seconds on average over all nodes with the method to be called 7630 times, resulting in a total time for the `xmp_read` method of 16.7112 seconds. The time we calculated for the FUSE overhead is 5.557 seconds. When subtracting these results and the 0.653 seconds for calculating the hash-sum from the total time, the remaining time of 0.004 seconds can be attributed to other factors. The `xmp_read` method of the Naive file system was also called 7630 times. The average time for reading the range of bytes was 0.00028, totaling 2.214 seconds. The broadcast operation took an average of 0.00175 seconds, totaling 13.355 seconds. Again, the time we calculated for the FUSE overhead is 5.557 seconds. Subtracting those values and the 0.653 seconds for the hash-sum from the total time, the remaining 0.305 seconds can be attributed to other factors. These results are visualized in Figure 4.

Unfortunately, only after these tests did we realize that a lot of performance was wasted by the very expansive metadata operations needed for the simple caching mechanism. Before, the location for every byte was stored in the metadata with the according node ID. The metadata buffer was used to determine on which node the needed data was located or if it was already stored locally. Such a high resolution on the meta resulted in vast amounts of required operations, especially when working with large files. To test how much performance was lost, another small set of tests was run without using a metadata buffer at all. Instead, the nodes where the requested data is located were inferred using the same algorithm to split the file and populate the metadata buffer in the `xmp_open` method. The results of these tests, compared to the old results, can be found in Figure 5. We can see that we save at least 5 seconds without the meta buffer operations. This means this version of the OSR file system is more efficient than the native home file system starting with only eight nodes.

VI. Discussion

On the Home and Scratch file systems with small file sizes, the custom file systems are slower than the native file systems. In the case of the broadcast-based file systems, this is caused by the MPI overhead, while for the OSR file system, this is caused by the `xmp_open` method and the MPI overhead. Compared to the One-Sided Reading design, the broadcast-based file systems don't show any performance loss with an increasing number of nodes here since the file can probably be broadcasted with only one operation. In contrast, the One-Sided Reading design must start transferring data between nodes in smaller batches. With increasing file sizes, the broadcast-based file systems get significantly worse with increasing numbers of nodes. The limiting factor here might be that the nodes must wait a significant amount of time between broadcast operations until all nodes have reached the next broadcast operation. That gets worse with more nodes and adds up with larger files, requiring a more significant number of broadcast operations. It is unexpected, though, that accessing the file directly with the Naive file system is so much slower with large files than copying the file first. That might be caused by extra waiting time when waiting for the next broadcast when some nodes need more time to process the received data inside the `sha256sum` algorithm than other nodes. The native Home file system starts to show its limitations with the 10 MB file already and has a maximum throughput of slightly above 1 GB/s. While the One-Sided Reading file system is significantly worse than the Home file system with a file size of up to 10 MB, it also reaches a maximum throughput of around 1 GB/s with the 1 GB file (see Table I). But more importantly, the scaling from one node to 24 nodes for the OSR file system (1.617) is better than for the Home file system (2.910) when accessing the 1 GB file. The hypothesis is that with just some more nodes (32 or more), the OSR file system might start to be faster than `/home` if both trends continue with a more significant number of nodes. This is likely because the performance of the Home file system is expected to worsen with an increasing number of nodes because of the bandwidth bottleneck. In contrast, we can expect the performance of the OSR file system to continue to grow linearly as the bandwidth of the communication channels between the nodes is not limiting the performance at this scale. It is clear that the MPI overhead limits the Naive design approach and OSR file system. The simple Passthrough shows a similar performance to the Home file system. The FUSE overhead is not so noticeable here since the Home file system's latency is already high.

It is also interesting to note that we have some unexpectedly high results; for example, with 10 MB and two nodes and with 1 GB and two nodes, the timings for most file systems are unexpectedly higher even compared to four nodes. That might be due to 'noisy neighbors' also stressing the network and storage nodes. Curiously, the OSR file system does not seem to be affected by it as much. That might be because it only reads the test file during the open method and relies on communication between the nodes afterward, thus not affected as much by 'noisy neighbors'. However, that is only speculation and is difficult to verify.

When looking at the results when accessing the file over the Scratch file system, the results are similar for the FUSE file systems. The Scratch file system itself proves to be much faster than Home. That can be due to various aspects, such as better network bandwidth over a fabric connection, multiple storage nodes sharing the load, better storage devices, etc. It is also notable that for all file sizes, the performance gets better with an increasing amount of nodes, which can probably be attributed to caching on the storage servers themselves. The results of the Passthrough file system show the overhead caused by FUSE, especially with the 1 GB file. The Passthrough file system is multiple seconds slower than Scratch, caused by the multiple context switches, as mentioned in Section II. This performance loss also contributes to the results of the other custom file systems. We also do not have unexpected spikes, as we observed when using the Home file system, since 'noisy neighbors' have less effect on our jobs when the native file system does not reach its bandwidth limit as easily.

The small remaining time for both file systems confirms that our measurements are accurate and that we identified the most critical performance factors. In the case of the Naive file system, the MPI overhead is associated with the time the broadcast operation takes in the `xmp_read` method (orange tile). For the OSR file system, the MPI overhead is associated with the entire time `xmp_read` method takes (orange tile) since the file is already entirely in memory, distributed over the nodes, and all operations in the `xmp_read` method are related to remotely accessing the file using MPI. The FUSE overhead is the same for both file systems since the FUSE overhead is dependent on the number of file system operations, which is not influenced by the file system itself. The FUSE overhead should also be the same for any number of nodes since the number of file system calls per node is unaffected by the number of nodes running the job. For both file systems, the FUSE overhead is significantly less than the time the MPI commutation takes, while the overhead caused by the MPI communication will increase with a growing number of nodes. Since this work aims to develop a file system that scales well over an increasing number of nodes, we can confidently say that the MPI communication overhead is the most significant factor for performance. Limiting the MPI overhead should be the first priority to improve the performance of the file system as a whole. This could be done by reducing the number of times the `MPI_Get` method is called to a minimum, for example, by always reading the entire chunk of the file that is assigned to a node so that each node never has to communicate with another node more than once. This would increase the number of times that a read can be covered by the local buffer and decrease the number of times the `MPI_Get` method is called.

After analyzing the results of the performance factor analysis, we noticed that a significant amount of time spent during the `xmp_read` method of the OSR file system is caused by a large number of metadata operations interacting with the metadata buffer. This was mainly due to the very high resolution of the metadata buffer, keeping track of the location of each byte. To quantify the resulting performance impact, an updated version of the file system was implemented without any metadata buffer, eliminating the metadata operations. Removing the buffers means we don't have any caching mechanism, which imposes a performance loss in most use cases. However, we wanted to test the performance without the caching mechanism since we suspected it to be extremely inefficient and impact the results more than it should. The resulting performance improvement is significant, with a 5-second improvement for all test configurations. When comparing the new results to the performance of the native file systems, the updated version exceeds the performance of the Home file system with only eight nodes. This performance gain is expected to grow with an increasing amount of nodes. The optimized Scratch file system is still much faster. However, we also did not create a workload that caused a bottleneck for this file system. It is important to note that the performance difference from the first version of the OSR file system depends on the size of the accessed file, not on the number of nodes accessing the file.

VII. Conclusion and Future Work

In conclusion, an MPI-based FUSE file system is presented that can use two different methodologies to orchestrate I/O from multiple nodes. For highly synchronous reading operations, a broadcast mechanism that could read a 1 GB file using less than 4% of the bandwidth compared to the baseline is shown. This entailed a performance penalty of approximately 250% during the synthetic benchmark. In real-world scenarios, this performance penalty might decrease if the available bandwidth is also shared with other users. Therefore, one achieves a bandwidth-reduction to latency-increase ratio of approximately 8.

The more flexible OSR mechanism reduced the performance penalty to around 8% while retaining the same bandwidth reduction, resulting in a bandwidth reduction to latency increase ratio of approximately 22. The refined version of the OSR file system improved performance even further. It was able to reduce latency compared to the baseline while also retaining the bandwidth reduction.

Future work should focus on testing the OSR file system on a larger scale. Some technical improvements should also be made, like developing an efficient caching mechanism, running the file system multi-threaded on each node, and adding the capability to handle multiple opened files simultaneously.

References

[1] C. E. Leiserson *et al.*, "There's plenty of room at the top: What will drive computer performance after moore's law?", *Science*, vol. 368, no. 6495, eaam9744, 2020. DOI: 10.1126/science.aam9744.

[2] S. Lang *et al.*, "I/o performance challenges at leadership scale", in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09, Portland, Oregon: Association for Computing Machinery, 2009, pp. 1–12, ISBN: 9781605587448. DOI: 10.1145/1654059.1654100.

[3] W. Frings *et al.*, "Massively parallel loading", in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13, Eugene, Oregon, USA: Association for Computing Machinery, 2013, pp. 389–398, ISBN: 9781450321303. DOI: 10.1145/2464996.2465020.

[4] M. A. Mollah *et al.*, "A comparative study of topology design approaches for hpc interconnects", in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2018, pp. 392–401. DOI: 10.1109/CCGRID.2018.00066.

[5] T. kernel development community, "The linux kernel documentation - fuse", [Online]. Available: https://www.kernel.org/doc/html/next/filesystems/fuse.html?highlight=fuse (visited on 10/24/2024).

[6] Google, "Fuse-archive repository", [Online]. Available: https://github.com/google/fuse-archive (visited on 10/24/2024).

[7] A. Rajgarhia and A. Gehani, "Performance and extension of user space file systems", in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10, Sierre, Switzerland: Association for Computing Machinery, 2010, pp. 206–213, ISBN: 9781605586397. DOI: 10.1145/1774088.1774130.

[8] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To FUSE or not to FUSE: Performance of User-Space file systems", in *15th USENIX Conference on File and Storage Technologies (FAST 17)*, Santa Clara, CA: USENIX Association, Feb. 2017, pp. 59–72, ISBN: 978-1-931971-36-2.

[9] N. Hjelm, "An evaluation of the one-sided performance in open mpi", in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI '16, Edinburgh, United Kingdom: Association for Computing Machinery, 2016, pp. 184–187, ISBN: 9781450342346. DOI: 10.1145/2966884.2966890.

[10] W. F. Godoy *et al.*, "Adios 2: The adaptable input output system. a framework for high-performance data management", *SoftwareX*, vol. 12, p. 100 561, 2020, ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2020.100561.

[11] F. Zakaria, T. R. W. Scogland, T. Gamblin, and C. Maltzahn, "Mapping out the hpc dependency chaos", in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022, pp. 1–12. DOI: 10.1109/SC41404.2022.00039.

[12] D. Zhao *et al.*, "Fusionfs: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems", in *2014 IEEE International Conference on Big Data (Big Data)*, 2014, pp. 61–70. DOI: 10.1109/BigData.2014.7004214.

[13] I. Peng, R. Pearce, and M. Gokhale, "On the memory underutilization: Exploring disaggregated memory on hpc systems", in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, pp. 183–190. DOI: 10.1109/SBAC-PAD49847.2020.00034.