2021 9th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)
Amity University, Noida, India. Sep 3-4, 2021

2021 9th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO) | 978-1-6654-1703-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICRITO51393.2021.9596080

Challenges and Opportunities for Unikernels in Machine Learning Inference

Aarush Ahuja
Bharati Vidyapeeth's College of Engineering
New Delhi, India
ahujaarush@gmail.com

Vanita Jain
Bharati Vidyapeeth's College of Engineering
New Delhi, India
vanita.jain@bharatividyapeeth.edu

***Abstract*—Machine Learning has become a value creator for many new and old businesses. However, efficient realworld machine learning deployments are still a challenge. Traditional Machine Learning deployments suffer from efficient resource utilization and achieving predictable latency. They cannot be treated in the same manner as other application server deployments. Unikernels are a method to specialize application deployment and performance to suit the needs of the application. Traditionally, building or porting applications to unikernels have been challenging. However, recent work has been into simplifying the development of unikernels. Real-world Unikernels as of now are only for specializing applications that run on the CPU. We survey machine learning practitioners and find out that the majority of machine learning practitioners are using the CPU for machine learning deployments, thus, creating an opportunity for unikernels to optimize the performance of these applications. We compare the architecture of two unikernels: nanos and Unikraft. We benchmarked scikit-learn, a popular machine library, inside a unikernel and found that it only offered a 1% advantage over a traditional deployment. However, our testing could not include more innovative systems like Unikraft due to their immaturity and inability to run machine learning libraries. We include a dependency analysis of three popular machine learning libraries Tensorflow Lite, PyTorch and ONNX, to help pave the way for building machine learning applications as Unikraft unikernels.**

***Index Terms*—virtualization, machine learning, unikernel, cloud computing**

I. Introduction

Machine Learning practitioners are deploying models to the scale of hundreds of thousands of inferences per second. New and old businesses are adopting machine learning to create immense value in various applications across industries. This has created new challenges for practitioners to solve across the technology stack, especially to improve the performance of real-time inference systems. Swayam[1] and Clockwork[2] are works which highlighting necessity of building resource effective machine learning deployments. Most production deployments of real-time latency-sensitive machine learning systems are running on top of virtual machines in public clouds like Amazon Web Services, Google Cloud Platform and Microsoft Azure. Challenges of building latency sensitive applications on the cloud for traditional applications are well-known [3][4].

Unikernels bring an innovative approach to specializing applications. Instead of building applications to be run on top of an existing operating system such as Linux or Windows virtualized by a hypervisor, Unikernels allow building ”library operating systems” with usually only a single address space our desired application runs as the guest operating system managed by the hypervisor. This approach to software development allows for more effective resource utilization and especially faster boot times for applications compared to a traditional deployment. Especially when paired with lightweight hypervisors such as LightVM [5], Firecracker [6] etc. they can have extremely effective resource utilization and high performance. Various applications of Unikernels have been studied such as in Serverless systems[7], operating system design [8], ETL [9] and Edge Computing [10].

A unikernel’s ability to build high performance applications running on the CPU allows us to consider the applicability of unikernels in real-time latency-sensitive machine learning deployments, especially for improving resource utilization or building more reactive and cost effective serverless machine learning systems via Unikernels. However, the challenges for building machine learning systems are lesser known. An industry focused machine learning system should support the common frameworks and libraries such as Tensorflow, PyTorch or a model runtime such as ONNX required by machine learning practitioners. We consider the case of two unikernels, Unikraft [11] and nanos [12] evaluating the challenges in using the aforementioned frameworks as a unikernel. We compare machine learning libraries as a unikernel and compare it with a traditional Linux userspace deployment.

However, it is also necessary to understand the applicability of Unikernel powered machine learning in real world systems. We survey machine learning practitioners from various organizations in India to identify the variety of deployments and inference hardware used. Many machine learning pipelines might not opt to use accelerator hardware such as GPUs or other external accelerator hardware for their deployments and instead only depend on the CPU for various reasons including but not limited to their organizational SLAs, high cost of accelerator hardware.

Section II reviews literature on machine learning deployments, unikernels and applications of unikernels. Section III discusses the nanos unikernel and the Unikraft unikernel. Section IV presents a survey of machine learning practitioners. Section V evaluates the performance of the scikit-learn library as a unikernel and presents a dependency analysis of PyTorch and Tensorflow Lite to understand how they can be integrated into the Unikraft framework. Section VI discusses future work that could be pursued in context of unikernel machine learning. Section VII summarizes the paper’s contribution.

978-1-6654-1703-7/21/$31.00 ©2021 IEEE

Authorized licensed use limited to: Politecnico di Milano. Downloaded on August 16,2025 at 08:37:48 UTC from IEEE Xplore. Restrictions apply.

II. LITERATURE REVIEWS

Paleyes et al. [13] identifies a machine learning system to be spread across four steps: Data Management, Model Learning, Model Verification and Model Deployment. All of these steps have their own challenges and considerations. Our primary goal is to study Model Deployment, this step faces challenges in integration of machine learning models across operational and development teams, monitoring of the deployed models and updating the models continuously. However, as the number of models deployed and rate of inferences by an organization grows, the need of efficient and reliable distribution has become important. Swayam [1] and Clockwork [2] are two systems designed to fulfil characteristics such as low-latency inference, predictable serving latency and efficiency of resources across a networked cluster of inference hardware such as CPUs or GPUs. Recent work in unikernels has been in two directions, building unikernels out of a specific programming language and its libraries such as OCaml in MirageOS [14] by adding OS functionalities to the applications, taking an existing OS and removing unnecessary functionality such as Rump [15] which re-engineers the BSD kernel and Lupine Linux [16] built on top of the Linux kernel these Unikernels are able to provide POSIX binary compatibility at the cost of high complexity still being left from the original OS in the unikernel. Unikernels provide the advantage of being extremely resource efficient and highly performant by eliminating the need for context switches, allowing to building optimizing I/O paths, optimizing OS functionality such as memory allocators depending on the application. Developing and porting applications to unikernels have been a complex job however Unikraft [11] presents a novel modular approach to simplify the development of unikernels by building a flexible microlibrary system on top of interfaces representing major OS functionality. Unikernels have been applied in various applications especially for high performance applications at the Edge [10], ETL systems [9] or serverless systems [7] taking advantage of unikernels to offer higher resource efficiency.

III. UNIKERNELS

A. Unikraft

Unikraft[11] is a modular and open source unikernel built around the concepts of micro-libraries and OS primitives represented via common interfaces. Operating system primitives such as memory allocators, schedulers, networks stacks and boot code in Unikraft have defined interfaces and are implemented by standalone micro-libraries. This approach increases modularity and promotes the idea of decreasing dependency amongst parts of the operating system. There are six different primitive interfaces in Unikraft for various parts of the kernel. A different standalone micro-library implements each interface. Instead of depending on a general-purpose system present in a monolithic kernel, Unikernels can be specialized based on applications requirements.

Unikraft provides an automated porting system which associates with the build system of existing applications, thus simplifying the process of building existing applications as unikernels. It builds upon this approach to provide an innovative POSIX compatibility layer which decreases the cost of system calls up to 60x by converting them into inexpensive function calls at linking time. Unikraft offers powerful debugging capabilities in the form of a remote GDB server and adding tracepoints via the ukdebug library for effective and detailed debugging information from the unikernel.

B. nanos

Nanos [12] is another unikernel which has a single process model like Unikraft. However, nanos focus on ELF and syscall compatibility rather than modularity as is the case in Unikraft. Nanos offers syscall compatibility via a run-time translation layer. This simplifies the development process as existing binaries can be packaged and executed as a unikernel but decreases the opportunities for optimization as the unikernel's functionalities are fixed to the functionality implemented by the developers of nanos. In comparison, Unikraft has a porting system which depends on the sources of the application and its build system. It takes considerably more effort in porting an application and its dependencies to Unikraft however because of statically linking the source code as a unikernel, Unikraft can remove the overhead of run-time system call translation and convert them into inexpensive function calls. Kuenzer et. al [11] highlights that run-time translation of system calls can be up to 20x more expensive than system calls implemented as regular function calls.

Nanos and Unikraft offers the advantages of Unikernels but with very different designs, implementations, and end-user experience. Table. I list the number of system calls offered by Linux 3.7 and the number of syscalls supported by the nanos and Unikraft unikernel. Kuenzer et. al [11] also highlights that the system calls supported by Unikraft fulfils the system call requirements of the top 30 Linux server applications.

IV. MACHINE LEARNING INFERENCE

The machine learning stack consists of various machine learning frameworks such as PyTorch, Tensorflow, model runtime frameworks such as ONNX, model compilers such as TVM[17] and compute backends such as CUDA, MKL DNN etc. Constituent libraries and frameworks of production machine learning systems are chosen or built as practitioners move through the stages of a machine learning project depending on the requirements of the stage. Training is a stage requiring much experimentation and thus it is convenient to use frameworks such as PyTorch and Tensorflow in this stage to increase productivity. However, these frameworks might not offer high performance, efficiency or portability that might be required for deploying a model in a real-time latency sensitive environments. Thus, to optimize the deployment of models frameworks such as TVM[17], a model compiler and ONNX a graph representation format for models.

TABLE I. NUMBER OF SYSTEM CALLS SUPPORTED IN UNIKERNELS AND NUMBER OF SYSTEM CALLS IN LINUX

Kernel	System Calls Supported
Linux 3.7	393
Unikraft	146
Nanos	148

Using these systems allow abstracting the actual execution backends of the models from the model representation. Different runtimes for executing a ONNX model can be

Authorized licensed use limited to: Politecnico di Milano. Downloaded on August 16,2025 at 08:37:48 UTC from IEEE Xplore. Restrictions apply.

implemented with different characteristics, as is the case with the ONNX Runtime which offers the same API for model execution with optimized hardware backends available for CPUs (using OpenMP, MKL DNN; across different architectures and operating systems such as x86, ARM and Windows, Linux, Mac etc. respectively), GPUs, and inference acceleration frameworks (such as OpenVINO or Windows TensorRT).

With various approaches of optimization available for model serving even on CPUs such as OpenMP, MKL

DNN. It is possible to build cost effective machine learning pipelines with only using the CPU for inference instead of employing expensive GPUs. Lind [18] et. al. highlights using GPUs for machine learning tasks is more useful when working with deep and complex neural neural network and the difference is not significant for simpler neural networks. Thus, there are technical reasons to choose CPUs for deployments instead of GPUs given that simpler models are to be deployed. We consider the approach of optimizing model inference on a CPU via specializing the application through as a Unikernel.

We also must understand the importance of CPU powered inference in context of the industry and its requirements. We conducted a survey with machine learning practitioners in India to understand the frameworks and environments of production machine learning systems. From the 20 responses in the survey, we get to know that most machine learning teams consist of teams of one to five people, PyTorch, Tensorflow, Spacy are the most common frameworks used with few opting to build their own frameworks and majority of the respondents deploy machine learning models on both CPUs and GPUs.

V. Unikernels and Machine Learning Frameworks

A. Machine Learning Performance

We evaluate the performance of the popular scikit-learn library as a unikernel. For this, we use one of the standard benchmarks offered by scikit-learn. The benchmark takes a vectorized dataset of various newsgroups, trains a classifier and tests it. We evaluate the performance of two classifiers: the ExtraTreesClassifier, a randomized decision tree-based classifier and the MultinomialNB classifier, a multinomial Naive Bayes classifier. We package the benchmark as a nanos unikernel running Python 3.6.7 and build it for an AWS VM target i.e., a Xen virtual machine. For comparison to a traditional setup, we provision an AWS EC2 instance running Ubuntu 18.04 and Python 3.6.7. Both experiments were conducted on an t2.micro AWS EC2 instance in the ap-south-1 region.

We take the case of scikit-learn and more classical models over deep learning models for our benchmark because through interaction with practitioners we learned that most deployments on the CPU are for classical models implemented in scikit-learn rather than deep learning models which can take better advantage of GPUs. The benchmark includes a training segment where the models are trained on the dataset and a testing segment where the models are tested with sample data from the dataset i.e., inference is performed with the trained model.

TABLE II. Scikit-Learn performance in nanos unikernel vs traditional Linux deployment for MultinomialNB

Benchmark	Unikernel	Linux	Relative (Unikernel/Linux)
Training	0.27s	0.25s	8.0%
Testing	0.039	0.038	2.6%

TABLE III. Scikit-Learn performance in nanos unikernel vs traditional Linux deployment for ExtraTreesClassifier

Benchmark	Unikernel	Linux	Relative (Unikernel/Linux)
Training	44.19s	43.58	1.3%
Testing	1.31s	1.32	0.7%

Table. III and Table. II highlight the training and testing performance for the MultinomialNB and ExtraTreesClassifier in the nanos kernel. We find out that for both the benchmarks we only see minimal advantage for the task of inference i.e., our required use case. For inference, the nanos unikernel can offer only a mere 1% average advantage. Thus, we can conclude that the nanos unikernels approach of binary compatibility via runtime translation system calls is not effective for optimizing machine learning systems.

In comparison, Unikraft offers a different approach where system calls are equivalent to function calls as described previously. However, due to the immaturity of the Unikraft framework as of yet we have not been able to benchmark Unikraft. We present a dependency analysis of three popular machine learning libraries: Tensorflow Lite, PyTorch and ONNX.

B. Challenges with Unikraft

We compare the build systems of different machine learning frameworks such as PyTorch, Tensorflow and ONNX and identify the challenges in fitting these frameworks in the Unikraft system. The Unikraft unikernel depends on the availability of library sources to link the build objects with the unikernel. Based on the external dependencies and operating system functionalities required by a library, Unikraft micro-libraries can be created to fulfil these dependencies and these micro libraries can be built on top of the existing libraries and unsupported parts be patched to support the Unikernel. A high-level library such as a machine learning framework depends on many external libraries. For Unikraft, the most apt target to be built into a Unikernel is a C implementation of the library as it will require the least number of dependencies.

We evaluated three popular machine learning frameworks: PyTorch, Tensorflow, and ONNX.

PyTorch offers a C++ distribution named libtorch which is the apt candidate for linking with the Unikraft porting framework. The Unikraft community has done initial work on porting PyTorch to Unikraft, however it is still a work-in-progress. Due to being implemented in C++, PyTorch requires the C++ standard library. Unikraft requires libraries libunwind, libcxx, libcxxabi, compiler-rt and a libc such as newlib to successfully build a C++ application such as PyTorch. To support methods involving model downloading, a network stack is also required which can be fulfilled by the lwip micro library. Apart from libraries described above which are essential to

Authorized licensed use limited to: Politecnico di Milano. Downloaded on August 16,2025 at 08:37:48 UTC from IEEE Xplore. Restrictions apply.

many applications, PyTorch also requires various scientific computation libraries which are important to machine learning frameworks such as gemmlowp, eigen, sleef, libfxdiv. PyTorch also uses the serialization library protobuf required by the Caffe2 library.

While PyTorch is a full-blown machine learning framework, ONNX is a portable graph format for machine learning models. As of now, The Unikraft community has done some initial work on its implementation. It requires a pthread implementation such as pthread-embedded, a libc implementation such as newlib and python to build the python bindings. Similar to PyTorch, onnx also requires protobuf to serialize the models.

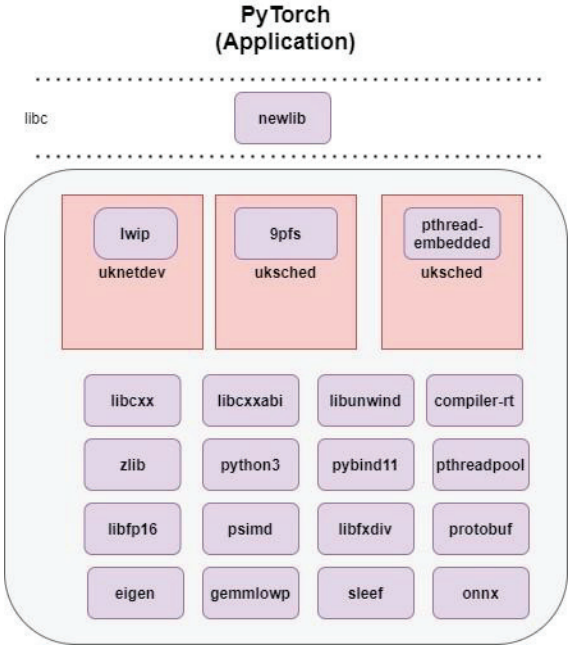

Fig. 1. PyTorch Dependency Analysis for Unikraft

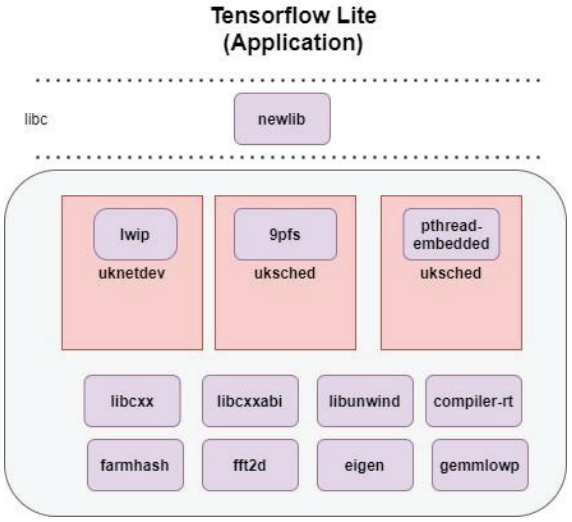

Fig. 2. Tensorflow Dependency Analysis for Unikraft

Tensorflow is also another framework like pytorch. There has been work done by the Unikraft community to build tensorflow-lite as a unikernel and expose its C++ API. The build requirements of tensorflow-lite are like that for PyTorch requiring a C++ runtime, a network stack, a thread scheduler and various scientific computation and math libraries such as gemmlowp, fft2d etc. Instead of using protobuf for serializing the models, Tensorflow uses flatbuffers.

To build existing libraries in a unikernel, many parts of these libraries must be patched or modified according to the features present in the unikernel. These patches can involve removing unnecessary functionality such as CUDA-specific code in different frameworks described above, modifying certain system specific features such as system calls or calls to external code etc.

Using this information, we can build the external libraries required by the machine learning frameworks as Unikraft micro-libraries. Once these dependencies are fulfilled, the machine learning frameworks can be patched and linked inside the Unikernel.

VI. Future Work

Unikernels are advantageous due to their low resource consumption, high density per host and fast boot times. These advantages can translate well into a production system for serverless machine learning building on top of systems such as SEUSS[7].

VII. Conclusion

We covered the background in how Unikernels work, considering two different unikernels: nanos and Unikraft having different architectural designs and how Unikraft innovates. We surveyed machine learning practitioners in India to understand the machine learning deployments in production across metrics such as CPU/GPU usage for inference, team size and top libraries and frameworks used. We found out that majority of machine learning teams in India consist of one to five people and deploy models on both CPUs and GPUs, thus proving the need for optimizing inference on CPUs. Our evaluation of scikitlearn inside a unikernel finds out that the nanos Unikernels with run-time translation of system calls did not offer any significant gains in performance over traditional deployments. We consider the applicability of Unikraft for powering machine learning inference evaluating three popular machine learning frameworks via analysis of their build systems and dependencies. This can help pave the path for development of these libraries as Unikraft unikernels. The future is bright for Unikernels, and we believe there are many opportunities to utilize the advantages of Unikernels especially for real-time serverless machine learning systems.

References

[1] Arpan Gujarati, Kathryn S Mckinley, Björn B Brandenburg, Sameh Elnikety, and Yuxiong He. Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services with Resource Efficiency. 17.

[2] Arpan Gujarati, Reza Karimi, Antoine Kaufmann, and Jonathan Mace. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. Technical report.

[3] Zhitao Wan. Cloud Computing infrastructure for latency sensitive applications. In International Conference on Communication Technology Proceedings, ICCT, pages 1399–1402, 2010.

Authorized licensed use limited to: Politecnico di Milano. Downloaded on August 16,2025 at 08:37:48 UTC from IEEE Xplore. Restrictions apply.

[4] Istvan Pelle, Janos Czentye, Janos Doka, and Balazs Sonkoly. Towards latency sensitive cloud native applications: A performance study on AWS. In IEEE International Conference on Cloud Computing, CLOUD, volume 2019-July, pages 272–280. IEEE Computer Society, jul 2019.

[5] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than your Container CCS CONCEPTS. 16, 2017.

[6] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pages 419–434, Santa Clara, CA, February 2020. USENIXAssociation.

[7] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: Skip Redundant Paths to Make Serverless Fast. page 15.

[8] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Pierre Olivier, Tiberiu Mosnoi, Răzvan Deaconescu, Felipe Huici, Costin Raiciu, and tefan Teodorescu. FlexOS: Making OS Isolation Flexible; FlexOS: Making OS Isolation Flexible. In Proceedings of the Workshop on Hot Topics in Operating Systems, New York, NY, USA. ACM.

[9] Henrique Fingler, Amogh Akshintala, and Christopher J Rossbach. USETL: Unikernels for Serverless Extract Transform and Load Why should you settle for less? 19.

[10] Shichao Chen and Mengchu Zhou. Evolving container to unikernel for edge computing and applications in process industry, feb 2021.

[11] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft. In Proceedings of the Sixteenth European Conference on Computer Systems, pages 376–394, New York, NY, USA, apr 2021. ACM.

[12] nanovms/nanos: A kernel designed to run one and only one application in a virtualized environment.

[13] Andrei Paleyes, Raoul-Gabriel Urma, and Neil D. Lawrence. Challenges in Deploying Machine Learning: a Survey of Case Studies. nov 2020.

[14] Anil Madhavapeddy and David J. Scott. Unikernels: The rise of the virtual library operating system. Communications of the ACM, 57(1):61–69, jan 2014.

[15] Antti Kantee. Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels. Technical report.

[16] Hsuan Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A Linux in unikernel clothing. In Proceedings of the 15th European Conference on Computer Systems, EuroSys 2020. Association for Computing Machinery, Inc, apr 2020.

[17] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[18] Eric Lind, Ävelin Pantigoso, Kth Skolan, För Elektroteknik, and Och Datavetenskap. A performance comparison between CPU and GPU in TensorFlow. Technical report.

Authorized licensed use limited to: Politecnico di Milano. Downloaded on August 16,2025 at 08:37:48 UTC from IEEE Xplore. Restrictions apply.