UNICORE: A toolkit to automatically build unikernels

Gain Gaulthier
University of Liege
Belgium
gaulthier.gain@uliege.be

Soldani Cyril
University of Liege
Belgium
cyril.soldani@uliege.be

Mathy Laurent
University of Liege
Belgium
laurent.mathy@uliege.be

ABSTRACT

Recent years have seen the IT industry move massively towards the use of virtualization for the deployment of applications. However, the two most prominent virtualization technologies, i.e. virtual machines (VMs) and containers, both present serious drawbacks. Full-blown VMs provide a good level of isolation, but are generally heavyweight. On the other hand, containers are generally more lightweight, but offer less isolation and thus a much greater attack surface.

Unikernels have been proposed to virtualize applications in a way that is both safe, and efficient. They are specialized operating systems, tailored for a specific application, which allows to build minimalist VMs with tiny memory footprints. They keep the increased security of VMs, but with performance equivalent to or even better than equivalent containers. Unfortunately, porting an application to the unikernel paradigm currently requires expert knowledge, and can be very time-consuming.

In this paper, we introduce UNICORE, a common code base and toolkit to automate the building of efficient unikernels from existing off-the-shelf applications. Although UNICORE is still in the early stages, we present early results showing that UNICORE images are able to yield performance similar or better than lightweight virtualization technologies such as containers.

CCS CONCEPTS

• **Software and its engineering → Virtual machines**.

KEYWORDS

Virtualization, unikernels, specialization, operating systems, Xen, containers, hypervisor, virtual machine.

1 INTRODUCTION

Cloud computing is becoming the core business of the IT industry. This trend is justified by the fact that network operators and service providers need high-performance services to deploy their networked applications to the market. When public clouds appeared, the basic technology used was hardware virtualization. In this paradigm, the virtual machine (VM) is defined as the standard unit of deployment. Each VM is represented as a self-contained computer, booting a standard OS kernel and then running a specific application (e.g., a database, a web server, ...). Running multiple VMs on the same machine significantly reduces costs. Although VMs strongly reduce the required number of physical machines, they introduce considerable drawbacks. Indeed, since they require a full operating system image to run (kernel and applications), they are heavyweight. Running many of them on the same hardware requires thus a lot of RAM and CPU cycles and impacts performance. Furthermore, due to their large size, they waste disk space and limit boot and shut down time.

These drawbacks lead the IT industry to embrace containers to replace virtual machines. This transition is intended to improve performance, speed-up software deployment and reduce costs. Rather than virtualizing the underlying hardware like VMs do, containers virtualize the OS itself, sharing the host OS kernel and its resources with both the host and other containers. This model considerably reduces the memory wasted by duplicating OS functionality across VMs and improves the overall performance.

In recent times, container-based technologies such as Docker [3] and LXC [8] have gained enormous traction. Large internet companies such as Google and Amazon decided to set aside virtual machines by running all of their services in containers [20]. The reasons of this major change are quite clear. In contrast to heavyweight VMs, containers provide fast instantiation times, small memory footprint and reduce size on disk.

Nevertheless, no technology is perfect, and containers are no exception. Indeed, as they share the host OS kernel and contains numerous binaries and libraries, the attack surface is large. They are thus subject to a lot of vulnerabilities [2]. In addition, the kernel system call API that containers use to interact with the host OS represents also a serious flaw and is the target of an increasing number of exploits [9].

At this point, software corporations face a dilemma between inherently insecure containers and heavyweight but isolated virtual machines. Could we do better by having a lightweight but isolated environment? Fortunately, it is possible and a new technique taking the best of two worlds exists.

Offering a great trade-off between performance and isolation, a new model has been designed to replace virtual machines and containers: unikernels. Also known as lightweight VMs, they are specialized VMs that include only the minimum feature(s) to run a specific application. Unikernels [23] are thus the smallest lightweight virtual machines that can be created. They can run directly on top on a hypervisor or bare metal, eliminating the need for a host operating system. In a unikernel, the application is compiled only with the necessary components of the operating system (e.g., memory allocators, schedulers, device buses, ...). Their size are thus considerably reduced, resulting in better performance and attack surface. Finally, because unikernels are minimalist, they are also easier to verify, not only for quality but also for safety and security. Many unikernels have been developed already such as ClickOS [13], LightVM [12], IncludeOS [1]... They all offer great performance and low memory footprint for their chosen task. Figure 1 illustrates the major differences between VMs, containers and unikernels.

The fundamental drawback of this paradigm is that it is necessary to manually port existing application(s) to the underlying unikernel. For example, a web server can be ported as a unikernel

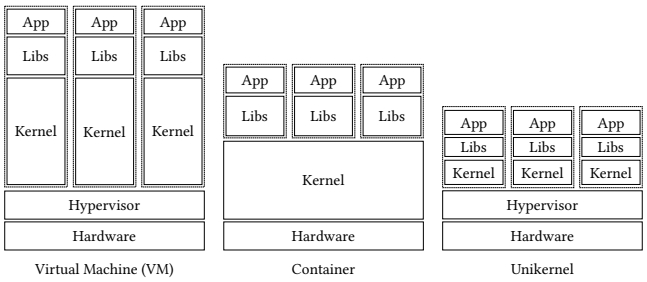

Figure 1: Comparison of virtual machine, container and unikernel system architecture.

by selecting and extracting the right operating system components and primitives (e.g., network stack, network drivers, ...) while respecting a given API. Porting legacy applications is in general not trivial. Indeed, several factors such as incompatible or missing libraries/features, complex build infrastructures, lack of developer tools (debuggers/profilers), and unsupported languages prevent unikernels to gain significant traction from developers. Furthermore, the migration represents only a small portion of the work. Indeed, other processes such as verification and optimization are necessary to obtain a fully operational and optimized unikernel for a particular platform and architecture. All these manual steps involve significant resources and complex operations which are tedious for developers. These challenges prevent unikernels from being widely used by the software industry.

To circumvent costly operations related to unikernel development and deployment, a new research project is being studied and developed: UNICORE. The main objective of UNICORE is to develop an open-source toolkit to automatically build minimalist operating systems targeting a single (or multiple) existing application(s) that is/are optimized to run on different architectures (e.g., x86, ARM, MIPS) and platforms (e.g., bare metal, KVM, Xen, ...). In this way, the resulting unikernel(s) will have small image size, fast boot time, and low amount of memory used.

The rest of this paper is organized as follows. Section 2 describes the unikernel architecture and discusses its general principles. Section 3 reviews related work. Section 4 introduces UNICORE, its main concepts and objectives. Section 5 gives an overview of our actual research and work on UNICORE. Section 6 presents early results. Section 7 discusses future work and any shortcomings that have been identified. Finally, Section 8 concludes the paper.

2 UNIKERNEL ARCHITECTURE

Unikernel is a relatively new concept in which an application is directly integrated with the underlying kernel. In other words, software is compiled against bits of OS functionality that it needs (e.g., only the required system calls and drivers) into an executable image using a single address space. Single address space means that in its core, the unikernel does not have separate user and kernel address spaces and all threads and the kernel use the same page table.

Therefore, unikernels can only run a single process at a time. As a result, they do not support forking. Nevertheless, they can fully support multi-threaded applications and multi-core VMs [5]. The first advantage of this approach concerns the reduced attack surface and exploitable operating system code. In contrast to virtual machines and containers which offer solutions that are packed with more tools and libraries than required by the running application, unikernels only contain the necessary operating system functions. Furthermore, such a paradigm allows also to exclude shells. Indeed, a number of attacks try to invoke a shell to alter the system they are attacking. Without a shell, an attacker does not have this opportunity. This forces the attacker to use machine code to subvert the system which decreases the chances of successfully completing the attack.

In addition to security, unikernels also allow to improve performance. They use a single address space, without distinction between kernel-space and user-space. Consequently, system calls become equivalent to regular function calls, avoiding the overhead of context switches and data copies between user and kernel spaces.

Single address space means running application in kernel mode. This model implies that software bugs will critically break the running unikernel. Due to this design, unikernels are also harder to debug since they usually do not provide their own sets of debugging tools. To use existing tools, it is necessary to cross-compile them. Additionally, third party libraries used by the debugging tools must be included into the image, ballooning the size of the unikernel. Any debugging tools based on multiple parallel processes can not work in a unikernel by design.

3 RELATED WORK

There exist plenty of related work showing that unikernels bring great benefits and impressive performance compared to traditional VMs and containers. It is possible to divide research-related area into two main categories: (1) *Development of minimalist operating systems that are POSIX-compliant.* These can run existing and legacy application by using cross-compiling techniques. Generally, they are based on a custom kernel and use a larger code base since they require more resources. Nevertheless, these platforms provide an easier way to migrate traditional software (running on virtual machines and containers) into unikernels, since the application only needs to be recompiled. OSv [5] is an example of this type of system. It is designed to run unmodified Linux applications on the KVM [6] hypervisor. In the same family, Rumprun [21] provides reusable kernel-quality components which allow to build highly customized images with minimal footprint. (2) *Development of minimalist operating systems with custom API.* Unlike the previous approach, this model does not try to optimize existing code, but instead focus on a set of tools to quickly assemble new components without having to deal with underlying services (e.g., memory allocators, drivers, ...). The downside of this concept is that it provides code base which are generally incompatible with existing applications. Therefore, they require to rewrite the legacy code using the defined platform's API. For example, MirageOS [11] written in OCaml, is based on this architecture. It is designed as a complete clean-slate set of protocol libraries with which to build specialized unikernels that run over the Xen [27] hypervisor.

Both approaches require significant expertise (often missing in application developer's toolkit) and development time to port an

existing application as a unikernel. The adversity of compiling unikernels may significantly hamper their widespread adoption by the software industry. Fortunately, the UNICORE project's objective is to change this status quo by providing a highly configurable unikernel code base being agnostic to the underlying hardware or virtualization technology.

4 UNICORE

The high-level goal of UNICORE is to be able to build unikernels targeted at specific applications without requiring the time-consuming, expert work that building such a unikernel currently requires. In order to achieve this objective, UNICORE will provide a code base with library pools (Section 4.1) as well as a toolkit (Section 4.2) comprising a set of tools to automatically build images of operating systems targeting a single (or multiple) applications that are optimized to run on bare metal or as virtual machines. UNICORE will thus reduce several critical metrics such as image size, boot time, and the amount of memory used and ensure strong isolation, and performance comparable to containers. The main concept of UNICORE is shown in Figure 2.

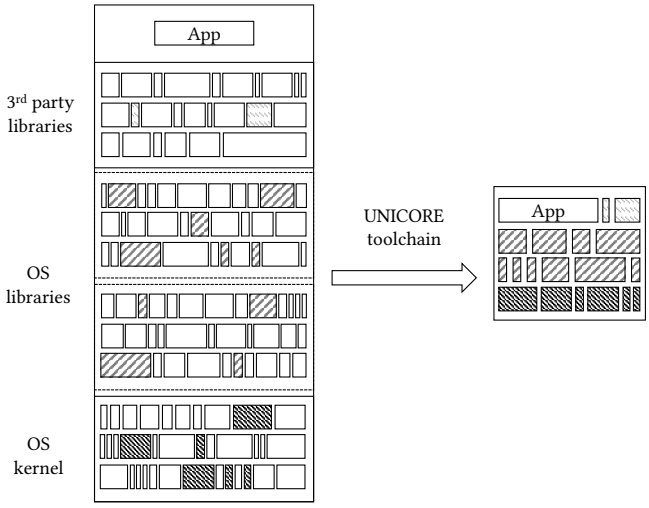

Figure 2: UNICORE high-level concept.

Since UNICORE will greatly ease the porting of existing applications, already contains an increasing number of supported libraries, and is fully open-source, we hope it will gain traction in the community, contrary to anterior attempts requiring more development effort [23].

4.1 Library pools

In order to develop fully specialized and customized unikernels, UNICORE will first decompose operating system primitives and libraries into fine-grained modules called micro libraries or μlibs. Such decomposition will allow to select and include only the necessary μlibs that applications require to run. Therefore, resulting images will be extremely thin and optimized. Using μlibs will also ensure modularity and code reusability by providing a common code base for unikernels development. UNICORE will thus provide pools of various libraries which can be selected to build unikernels. These pools of libraries thus constitute a code base for creating unikernels. UNICORE libraries are divided into three pools illustrated in Figure 3: (1) *Main lib pool* contains libraries that provide basic pieces of functionality. Libraries can be arbitrarily small or as large as standard libraries like libc. These libraries are themselves divided into internal and external libraries: *Internal lib pool* provides functionality typically found in operating systems (e.g., memory allocators, schedulers, device buses, ...) and are part of the UNICORE core. *External lib pool* consists of existing software projects external to UNICORE. For example, these include libraries such as musl [14], but also language environments such as Golang [4] and Javascript/v8 [25]. (2) *Platform lib pool* contains all libraries for a particular target platform such as Xen, KVM and bare metal. (3) *Architecture lib pool* provides libraries dedicated to a specific computer architecture (e.g., x86, ARM, MIPS, ...). In addition to these categories there exist also applications that can be ported to the UNICORE project. They correspond to standard applications such as MySQL [15], Nginx [16] or PyTorch [19], to name a few.

One important thing to point out regarding internal libraries is that for each category, UNICORE defines (or will define) an API that each library under that category must comply with. In this way, it is possible to easily plug and play different libraries of a certain type (e.g., using a cooperative scheduler or a preemptive one). An API consists of a header file defining the actual API as well as an implementation of some generic/compatibility functions, if any, that are common to all libraries under a specific category.

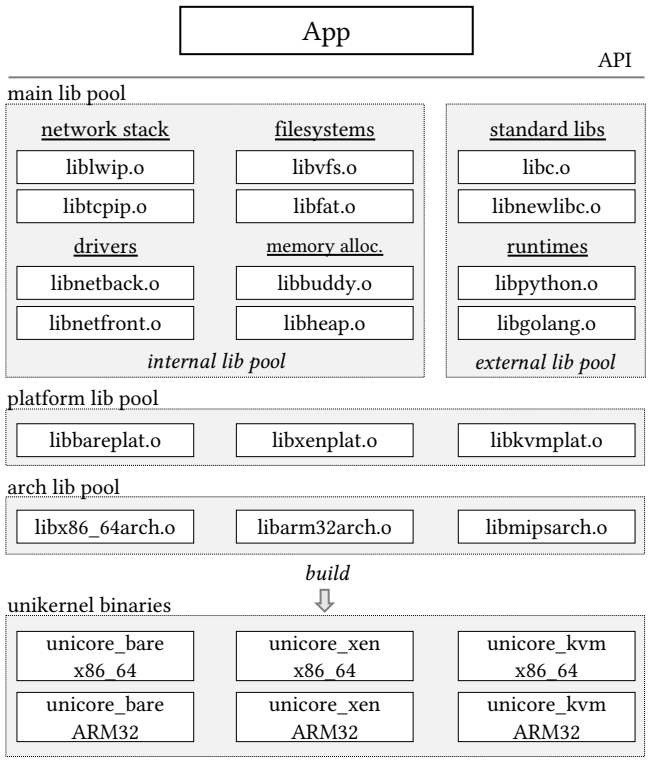

Figure 3: The UNICORE library pools.

4.2 Toolchain

The UNICORE toolchain will provide a set of tools to automatically build images of operating systems targeting applications. In a general way, the toolchain will build unikernels by extracting OS primitives and selecting the right micro libraries and third party libraries. In order to be able to build secure and reliable unikernels, several tools will be integrated to the toolchain: (1) *Decomposition tool* to assist developers in breaking existing monolithic software into smaller components. (2) *Dependency analysis tool* to analyze existing, unmodified applications to determine which set of libraries and OS primitives are absolutely necessary for correct execution. (3) *Automatic build tool* to match the requirements derived by the dependency analysis tool to the available libraries. (4) *Verification tool* to ensure that the functionality of the resulting, specialized OS+application matches that of the application running on a standard OS. (5) *Performance optimization tool* to analyze the running specialized OS+application and to use this information as input to generate even more optimized images.

The combination of these tools is shown in Figure 4 and represents a unikernel toolchain able to automatically build efficient, customized, verifiable and secure specialized operating systems and virtual machine images. The toolkit will ensure that applications become independent of the platforms and the architectures against which they are built.

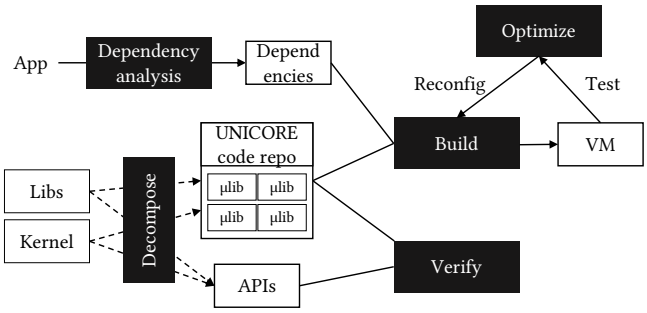

Figure 4: UNICORE tool flow during application construction.

Figure 4 represents the workflow for users of UNICORE. First, the dependency analysis tool will examine and extract dependencies from existing and unmodified applications. Then, the automatic build tool will use the dependencies as input to select the relevant micro libraries (see Section 4.1) from a library pools, link them against the application, and produce a minimalist UNICORE image for one or multiple target platform(s) and architecture(s). This image can then be deployed using standard provisioning tools such as OpenStack [18]. In parallel, the verification tool will ensure that μlibs are correctly implemented and that the newly built application is equivalent to the initial one. Finally, the resulting unikernel will be automatically profiled and the configuration parameters for the μlibs updated to improve performance, in an iterative process driven by the optimization tool.

5 CURRENT STATUS

At the time of writing, the project is still at an early stage. A first version of the code base as well as some libraries have already been written. Similarly, a first prototype of the toolchain has been developed.

Concerning code base and library pool, it was decided to follow the Unikraft [24] work. Unikraft includes some base libraries and relies on two existing tiny operating systems to support respectively the Xen and KVM hypervisors. These lightweight OSes interact with the hardware abstraction exposed by the hypervisor and form platforms for building language runtime environments and applications. Xen comes with MiniOS [28], a toy guest operating system which implements all of the basic functionality needed to run a lightweight image on Xen. This OS has a single address space, no kernel/user space separation, and a cooperative scheduler. To support KVM, we based ourselves on Solo5 [26]. In addition to Xen and KVM platforms, we also support the Linux userspace platform especially for debugging purpose.

A decomposition work has then been performed. As explained previously, UNICORE is based on a simple concept: everything is a library. Existing monolithic software was thus broken down into micro libraries according to their respective platform and architecture. For instance, memory allocator libraries and boot management libraries have been developed, providing low-level APIs to manage hardware abstraction. Subsequently, higher-level libraries have been implemented using underlying APIs. For instance, minimalist implementation of libc functionality such as nolibc but also larger micro libraries like newlib [22] or lwip [10].

In parallel to μlibs development, we are also working on the toolchain and two of its tool can already be used: the dependency analysis tool and the automatic build tool.

The first one aims to break down a binary application and analyses its symbols, its system calls and its dependencies. The tool relies on static and dynamic analysis and has been designed to examine binary files. Indeed, source code is not always accessible. Moreover, this approach allows to remain completely independent of the programming language. Finally, gathering data of binary is also useful for binary rewriting techniques [17], i.e., changing the semantics of a program without having the source code.

First, a static analysis (which does not require program execution) is performed on the binary file of the application. By disassembling the application and analyzing its assembly code, we can recover various information. To recover all functions and system calls, we examine and parse the ELF's symbols table[1]. Similarly, it is possible to recover shared libraries from a dynamic executable by parsing program headers and the `dynamic` section. Although, we obtain interesting results, this approach can have some flaws. Indeed, results of this analysis can be limited if binaries are stripped or obfuscated. Furthermore, some shared libraries can be omitted if they are dynamically loaded by the `dlopen` function. A second analysis is thus performed on the application. This one is dynamic since it requires running the program. It allows to collect additional entries on previous data (system calls, library calls and shared libraries). The optimal approach to gather all symbols and dependencies is

[1] An ELF's symbol table holds information needed to locate and relocate a program's symbolic definitions and symbolic references.

to explore all possible execution paths of the running application. Nevertheless, exploring all execution paths is in general infeasible. Fortunately, we can approach this scenario, by using a heuristic approach which requires high application code coverage. Tests with expected input and fuzz-testing techniques have been used in order to find the most possible symbols, system calls and dependencies. To gather symbols and system calls, we wrote several configuration files which contain various configurations to test (e.g., different ports numbers for web servers, background mode, ...) but also test files (e.g., SQL queries for database servers, DNS queries for DNS servers, ...). These configuration and tests files are then provided as input to the analyzer which monitors application's output by using the `strace`, `ltrace` and `lsof` [7] utilities. Once the dynamic analyzer finished its task, its results are compared and added to the ones of the static analyzer. Finally, all data is persisted on the disk and is used as input by the automatic build tool.

The automatic build tool is also divided into two subcomponents: a controller that uses shared libraries (gathered from the dependency analysis process) to select the right micro libraries (from library pools), and a build system to compile and link the unikernel into target VM images.

The controller subsystem performs a first matching based on shared libraries and μlibs nomenclatures. A second check is carried out based on symbols' names. In order to accomplish this task, each micro library publicly exports[2] its symbols into a special file. For each micro library, the controller reads this special file from the library pool, saves symbols and then updates (if necessary) the matching. Once this process is done, it calls the internal build system. This component is in charge of compiling the application and the selected libraries together to create a binary for a specific platform and architecture. The tool is currently inspired by Linux's kconfig system and consists of a set of Makefiles.

6 EARLY RESULTS

It is possible to present early results in terms of memory sizes and boot up times of UNICORE images. We developed a very simple unikernel that only prints "Hello world!" and then exits. This unikernel depends on several μlibs from library pools and is build with the automatic build tool. As shown in Figure 5, we build[3] three (uncompressed) images on different virtualization technologies: Xen, KVM and Linux userspace. Once built, we parse each resulting images (more specially the symbols table) and figure out which library contributed with which symbol(s).

As it can be observed, resulting images are extremely small. Unikernel for the Xen platform has a size of only 27 kB. Images for KVM and Linux userspace are respectively 34 kB and 14 kB. Bigger μlibs are related to platforms and nolibc. This is justified by the fact that platform related libraries contain all code needed to run a unikernel on a particular underlying platform. Similarly, nolibc uses a very minimalist libc implementation which contains a higher number of symbols than other μlibs such as memory allocator or time conversion μlibs.

[2]By default, there are separate namespaces for each library. In other words, every function and variable will only be visible internally.
[3]Built on: Debian 9; Linux 4.19.0; Intel®Xeon®E3-1231v3 3.40 GHz, 4 MB RAM for guests.

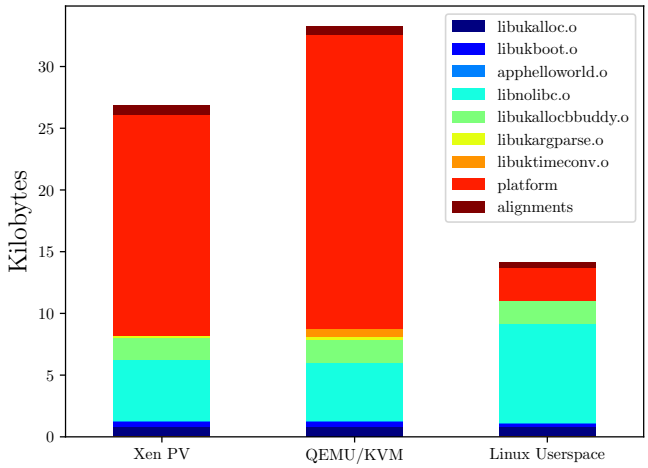

Figure 5: Contribution of minimal micro libraries set to *.text*, *.rodata* and *.data* sections for a simple "Hello World!" guest (uncompressed).

We also measure the total life cycle (create, boot, run, shutdown, and destroy times) of this same program on different platforms. As represented on Figure 6, the Linux userspace image is the fastest (only 0.32 ms) since it does not rely on any hypervisor. The total life cycle of Xen and KVM images is respectively 502.34 ms and 615.46 ms.

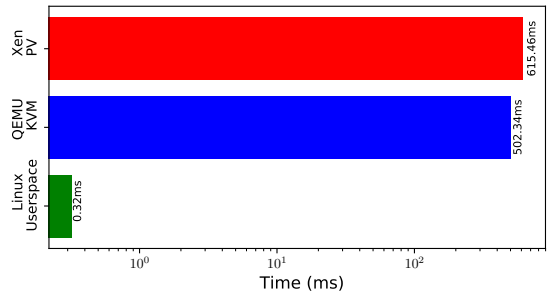

Figure 6: Total time to create, boot, shutdown, and destroy the "Hello world!" guest on different platforms.

These results show that UNICORE images have impressive performance compared to traditional VMs. For instance, a 1 GB Debian virtual machine has a total life cycle of several seconds. Compared to containers, the difference tends to be less pronounced. Indeed, since containers share the underlying kernel, their total life cycle can also reach hundred of milliseconds [12].

7 FUTURE WORK

As mentioned UNICORE is in analysis and development stages. There is a considerable work to be done concerning scientific research and implementation. Both on micro libraries development, APIs and toolchain design.

Although a large portion of micro libraries development concerns software engineering, there exist many research topics to consider and that can be exploited. Particularly, about how processes work. A traditional UNIX program uses specific primitives such as `fork` and `exec` to support multi-processing. In the context of unikernels, the classical paradigm of processes should be adapted. Indeed, unikernels are dedicated to run only a single application at a time and thus strip off the process abstraction from its monolithic appliance. In the same way inter-process communication can no longer be done using the traditional IPC. Can we update the classic paradigm of process creation by using a thread model? Does this design have a significant impact on performance? This is just one of the many possible areas of research for micro libraries. Indeed, by working at this low-level of abstraction, we can redefine mostly everything.

Concerning the toolchain, it is also possible to consider many research aspects. For instance, we can try to automatically decomposing existing software into working components and μlibs. In order to obtain lighter and more secure images, additional research on dead-code elimination must be performed. In addition, the verification tool represents also an important topic of research. Indeed, we must be able to ensure the correctness and the security of unikernels and μlibs. Finally, we will investigate the use of machine learning techniques to identify a set of guest OS and virtualization parameters and automatically test the given application with different parameters (e.g., how big should the page size be, ...) in order to obtain best performance.

8 CONCLUSION

We have taken a look at the exciting new paradigm of unikernels. While they have great potential in terms of performance, tiny boot times and small memory consumption, developing and deploying unikernels requires significant expert resources. Without an specialized infrastructure, they are not ready to be widely used by the software industry.

Fortunately, we are changing this status quo with a first prototype of UNICORE, a common code base and toolkit to deploy secure and reliable virtual execution environments for existing applications. UNICORE relies on two main components: library pools which provide a common code base for unikernels, ensuring a large degree of code reusability and a toolchain to automatically build and optimize unikernels.

Although UNICORE is at an early stage, we present early results in terms of memory sizes and boot up times of UNICORE images showing impressive numbers that cannot be reached by virtual machines and even containers.

ACKNOWLEDGMENTS

This work received funding from the European Union's Horizon 2020 Framework Programme under agreement no. 825377.

REFERENCES

[1] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E. Engelstad, and Kyrre Begnum. 2016. IncludeOS: A minimal, resource efficient unikernel for cloud services. *IEEE 7th International Conference on Cloud Computing Technology and Science, CloudCom 2015* (2016), 250–257. https://doi.org/10.1109/CloudCom.2015.89

[2] CVE Details 2018. Docker: Security Vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-13534/Docker.html

[3] Docker [n. d.]. The Docker Containerization Platform. https://www.docker.com/

[4] Golang [n. d.]. The Go Programming Language. https://golang.org

[5] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv—Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 61–72. https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity

[6] KVM [n. d.]. Kernel Virtual Machine. https://www.linux-kvm.org/page/Main_Page

[7] Linux Man page [n. d.]. Index of /linux/man-pages/man1. http://man7.org/linux/man-pages/man1/

[8] LinuxContainers.org [n. d.]. Linux Containers. https://linuxcontainers.org

[9] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[10] lwIP [n. d.]. lwIP - A Lightweight TCP/IP stack. https://savannah.nongnu.org/projects/lwip/

[11] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems - ASPLOS'13* 48, 4 (2013), 461. https://doi.org/10.1145/2451116.2451167

[12] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles - SOSP '17*. 218–233. https://doi.org/10.1145/3132747.3132763

[13] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. (2014), 459–473. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins

[14] Musl [n. d.]. musl libc. https://www.musl-libc.org

[15] MySQL [n. d.]. MySQL. https://www.mysql.com/

[16] Nginx [n. d.]. NGINX High Performance Load Balancer, Web Server, and Reverse Proxy. https://www.nginx.com

[17] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-Compatible Unikernel. (2019), 15.

[18] Openstack [n. d.]. Build the future of Open Infrastructure. https://www.openstack.org

[19] PyTorch [n. d.]. PyTorch: from research to production. https://pytorch.org

[20] RightScale 2018. State of the Cloud Report. http://assets.rightscale.com/uploads/pdfs/RightScale-2018-State-of-the-Cloud-Report.pdf

[21] Rumprun [n. d.]. The Rumprun unikernel and toolchain for various platforms. https://github.com/rumpkernel/rumprun

[22] The Newlib Homepage [n. d.]. The Newlib Homepage. https://sourceware.org/newlib/

[23] Unikernel.org [n. d.]. Unikernels: Rethinking Cloud Infrastructure. http://unikernel.org/projects/

[24] Unikraft [n. d.]. Unikraft - Xen Project. https://xenproject.org/developers/teams/unikraft/

[25] V8 JavaScript engine [n. d.]. V8 JavaScript engine. https://v8.dev

[26] D. Williams and M. Lucina. [n. d.]. Solo5. https://github.com/Solo5/solo5

[27] Xen Project [n. d.]. The Hypervisor (x86 and ARM). https://xenproject.org/developers/teams/xen-hypervisor/

[28] Xen Project [n. d.]. Mini-OS, Xen Project. https://wiki.xenproject.org/wiki/Mini-OS