Unikraft and The Coming of Age of Unikernels

Hugo Lefeuvre Gaulthier Gain Daniel Dinca
Alexander Jung Simon Kuenzer Vlad-Andrei Bădoiu
Răzvan Deaconescu Laurent Mathy Costin Raiciu
Pierre Olivier Felipe Huici

Article shepherded by: Rik Farrow

Thanks to their excellent performance, unikernels have always had a great deal of potential for revolutionizing the efficiency of virtualization and cloud deployments. However, after many years and several projects, unikernels, for the most part, have not seen significant, real-world deployment. In this article we argue that several factors contributed to this in the past, including lack of POSIX compatibility and the resulting lack of support for applications and languages, difficult or not widely adopted tooling ecosystems, lack of basic security features, and sometimes less-than-stellar performance. After many years of work on the Linux Foundation's Unikraft project, whose explicit goal is to tackle these issues directly, we believe that the time for unikernels to finally enter the main stage is now.

Introduction

Unikernels [7] have always had great promise: high performance (sometimes even higher than Linux), lightweightness in the form of incredibly fast boot times (a few milliseconds) and severely reduced memory usage, as well as strong security benefits, to name a few metrics. But why hasn't all of this potential materialized into wide use and deployment? We argue that in the past, four main reasons have hampered unikernels from becoming more widespread:

- **POSIX Compatibility**: Ultimately operating systems (and unikernels of course) are only as good as the applications they can run. Arguably, wide adoption depends solely on how good OSes are at running the applications and languages that people are interested in; for the most part, in the past, unikernels have had no or rather limited POSIX compatibility [10] (much more on this in Section *Application Compatibility: System Calls Support*).
- **Tooling Ecosystem**: Previous unikernel projects had little or no tooling ecosystems to improve usability. Those that did developed their own tools [4],

partly because there hasn't always been a clear de-facto standard as is the case now (e.g., Kubernetes for deployment). Asking potential users to adopt a new tooling ecosystem, or worse, having no such tooling, is always a tough ask and has been an obstacle to adoption.

- **Modularity**: In order to maximize lightweightness and performance through specialization, the unikernel should be fully modular. In the past, unikernel projects such as OSv and MirageOS [4, 7] relied on smaller, but still monolithic, kernels.

- **Security**: A few years back there were rather overblown claims about unikernels' strong security [1]. Although unikernels do have some intrinsic features that could *potentially* make them more secure (e.g., a very small Trusted Computing Base, immutability, no console, etc.), unfortunately most past implementations have lacked basic security mechanisms such as stack protector and ASLR [8].

We argue that after many years of struggle and false starts, unikernels, and in particular the Unikraft [5] Linux Foundation project (`www.unikraft.org`) are coming of age: its fully modular design allows for extreme specialization (and thus performance and lightweightness), standard security features such as stack protector are in place, and in the past months we have been putting effort towards seamless integration with Kubernetes and Prometheus, arguably the de-facto standards for deployment and monitoring. We give a high level overview of Unikraft in Section *Unikraft: a Modern, Fully Modular Unikernel*, focusing on its high degree of modularity and the resulting performance/lightweightness benefits.

What about POSIX compatibility? While Linux has over 300 system calls, previous studies [11] have shown through static analysis that only a subset (224) are needed to run a Ubuntu installation. This number is actually an overestimation due to various reasons, including the fact that not all such applications make sense in a unikernel context (e.g., desktop applications) and the imprecision of static analysis. In Section *Application Compatibility: System Calls Support* we will show a thorough investigation of what's actually needed to explain why Unikraft's 160 implemented syscalls (and counting) are more than enough to run a wide spectrum of applications, including Redis, SQLite, nginx, HAProxy, TFLite and Memcached, and languages like Python, Ruby and Go, to name a few.

Unikraft: a Modern, Fully Modular Unikernel

Unikraft is a novel micro-library OS that (1) fully modularizes OS primitives so that it is easy to customize the unikernel and include only relevant components and (2) exposes a set of composable, performance-oriented APIs in order to make it easy for developers to obtain high performance.

Figure 1 shows Unikraft's architecture. All components are micro-libraries that have their own Makefile and Kconfig configuration files, and so can be added to the unikernel build independently of each other. APIs are also micro-libraries that

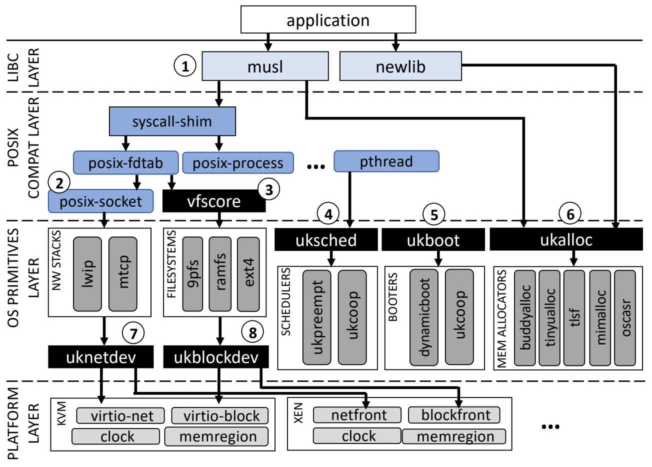

Figure 1: The Unikraft architecture enables specialization by allowing apps to plug into APIs (black boxes) at different levels and to choose from multiple API implementations.

can be easily enabled or disabled via a Kconfig menu; unikernels can thus compose which APIs to choose to best cater to an application's need.

Using Unikraft's build system and menu, it is quite easy to put together unikernels for a wide range of standard, off-the-shelf applications, and to automatically get the substantial benefits of deploying a unikernel without the hassle that was typical of past projects. Figure 2, for instance, shows memory usage for nginx, Redis and SQLite of only a few MBs compared to tens of MBs for even the leanest of Linux distributions; and Figure 3 shows Unikraft rates of 290K requests/second when running nginx on a single core on an inexpensive (less than $1000) server, clearly out-pacing Linux.

Clearly we don't expect all users to be interested in toying with Unikraft's build system and menu; for most users we provide pre-built images at `https://releases.unikraft.org/` as well as `kraft`, a wrapper tool that allows users to build unikernels with a single command (see `https://github.com/unikraft/kraft`).

Application Compatibility: System Calls Support

The ability to run a wide range of applications and languages is paramount to how much deployment any unikernel project will see. Unikraft addresses this directly through what we term *autoporting*: we use an application's native build system to

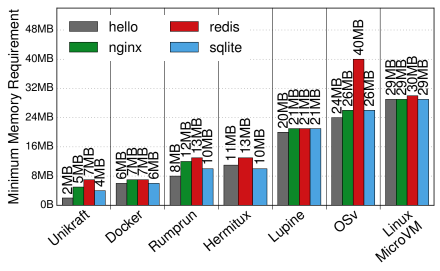

Figure 2: Minimum memory needed to run different applications using different OSes, including Unikraft.

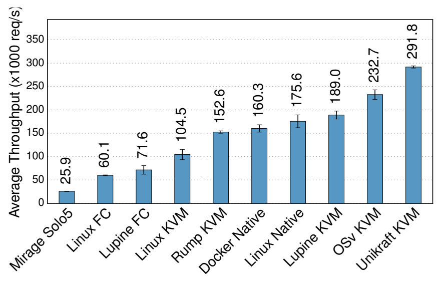

Figure 3: nginx and (Mirage HTTP-reply) performance with `wrk` (1 minute, 14 threads, 30 conns., static 612B page)

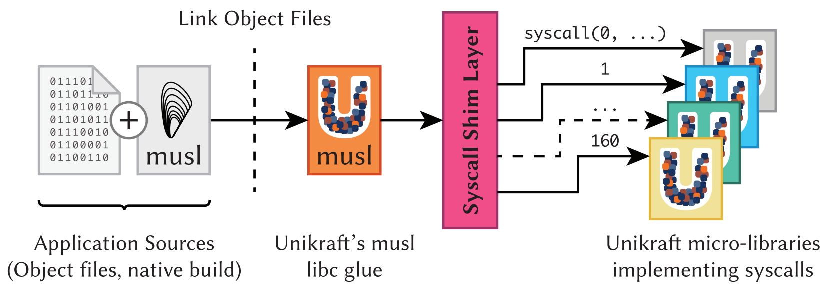

Figure 4: Unikraft avoids porting of applications by building them using their native build systems with musl and linking the object files into the Unikraft build.

build the application against the musl C standard library [9] and link the resulting object files against Unikraft (see Figure 4). For this to work, Unikraft includes a port of the musl library, which means, since musl is meant for Linux, that system call support is required. To address this, Unikraft provides (a modular) system call shim layer along with implementations of a growing number of syscalls (160 as of this writing). Unikraft also supports *binary compatibility* mode, where unmodified ELFs can be run on top of Unikraft with a slight performance hit; this functionality will be upstreamed in the future.

But is 160 enough, given that Linux has over 300 syscalls? We answer that question in the affirmative next.

A Trip Down POSIX-Compatibility Lane As a POSIX-like unikernel [4, 3, 10, 6], Unikraft strives for a high degree of compatibility with existing applications by supporting the Linux system call API. Some system calls are more popular than others [11] and the degree of compatibility of a given POSIX-like unikernel cannot simply be measured as the percentage of the Linux system call API it supports.

In a 2016 paper [11], Tsai et al. measured system call usage over the entire set of applications from a typical Ubuntu/Debian distribution. They concluded that to support 100% of these applications, 272 system calls needed to be implemented. That number went down to 202, 145, 81, and 40 system calls for the 90%, 50%, 10% and 1% most popular applications, respectively, suggesting that a large implementation effort would be required for an OS aiming at supporting even a small number of applications.

However, in the process of implementing POSIX system calls and checking whether applications where actually running, we gathered anecdotal evidence that these requirements were not as stringent as they seemed: whenever a system call is missing, the default behavior of Unikraft's system call shim layer is to stub it by returning `ENOSYS`; the result of this was that some applications where correctly running despite not having some of the system calls they invoked implemented, so we decided to take a closer look.

Looking Under the System Call API Hood To understand this behavior better, we performed a study that uses both dynamic as well as source-level static analysis. For the dynamic analysis we rely on `seccomp` to hook into each system call made by the application and to selectively disable it, returning either `-ENOSYS` (stubbed) or a success code even though the system call isn't actually implemented (faked). By monitoring the success/failure of the traced application, we can determine which system calls can be stubbed and/or faked. Further, we developed a static analysis tool that takes as input the application's and dependencies' code (e.g., libc) and outputs an estimation of the system calls the application may invoke at runtime. As a baseline we also ran the binary-level static analysis tools used by Tsai et al. [11].

For this initial analysis we focused on 5 applications (Redis, Nginx, Memcached, SQLite and HAProxy), although we are in the process of adding many more to the tool. We selected these because they are (a) popular applications (b) good candidates for running as a unikernel/cloud environment and (c) they have thorough benchmarking tools (`redis-benchmark`, `wrk`, etc.) and test suites. We use the benchmarking tools to provide realistic workloads and the test suites good coverage as we measure which syscalls the applications are making actual use of.

Results and Insights Figure 5 shows, for each application and corresponding benchmarking tool and test suite, the number of system calls statically identified and dynamically traced. Traced system calls are broken down between the ones whose implementation is absolutely required for the application/workload, as well as the ones that can be stubbed and/or faked.

The key insight is that applications are resilient to *a significant portion* of syscalls being stubbed and faked, and that the number of implemented syscalls they require to correctly run is significantly lower than the output of the static analysis suggests, let alone the total number of syscalls in the Linux API.

To confirm this, we took a look at the applications' source code: in cases where the failure of a system call is non-critical for the execution of the program,

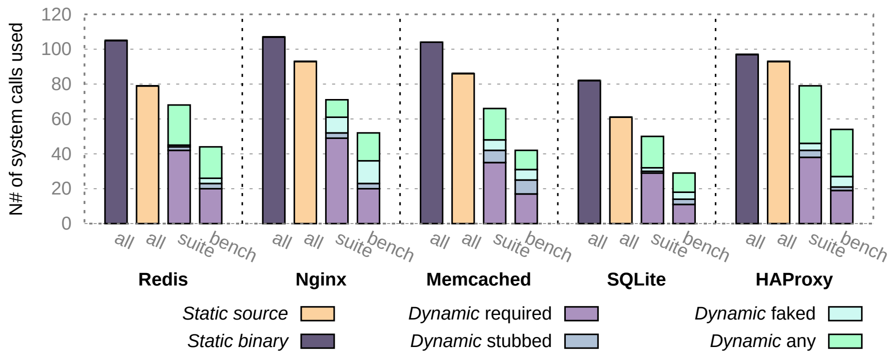

Figure 5: Number of system calls statically identified and dynamically traced for applications under traditional (*bench*) and unit tests (*suite*) workloads. Traced system calls are broken down into the ones that can be stubbed and/or faked and the required ones. Applications require much fewer system calls to run than a static analysis would suggest, and much less than the total syscalls in the Linux API.

the program can detect the error and decide to continue as usual, in which case stubbing works. This snippet from the Redis code-base is a good example:

```
if (getrlimit(RLIMIT_NOFILE,&limit) == -1) {
    serverLog(LL_WARNING,"Unable to obtain the current NOFILE"
        "limit (%s), assuming 1024 and setting the max clients"
        "configuration accordingly.", strerror(errno));
    server.maxclients = 1024-CONFIG_MIN_RESERVED_FDS;
}
```

Here when we stub `getrlimit` or `prlimit64`, Redis handles it gracefully with a default value. In other cases however, the program can interpret the error code conservatively and decide to abort, in which case faking usually works (since the failure of the system call is, in reality, non-critical). This snippet from the nginx code-base is a good example of such behavior:

```
if (prctl(PR_SET_KEEPCAPS, 1, 0, 0, 0) == -1) {
    ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
                  "prctl(PR_SET_KEEPCAPS, 1) failed");
    /* fatal */
    exit(2);
}
```

Here `prctl` fails to force the retaining of capabilities upon UID transition; in the context of an OS that does not have a user/kernel separation (as is the case for unikernels), capabilities make little sense and so it is fine to fake the success of the check.

On average, the proportion of invoked system calls that can be stubbed/faked is 42% for test suites, and 60% for traditional workloads. This shows that the effort to provide comprehensive (test suite level) support for these popular applications is relatively limited, and is even lower when considering partial support i.e. traditional workloads. We observe that the system calls that can be stubbed/faked vary among applications. As an indication, the number of system calls that would need to be effectively implemented for all these 5 applications to be supported is 78 for test suites and only 37 for traditional benchmarks.

A second observation is that static analysis produces many false positives and as such yields a large overestimation of the system calls made by an application. For example on Redis, binary-level static analysis identifies 89 system calls, vs. 68 dynamically traced from the test suite. These trends are the same for all programs. This is due to multiple reasons [2] such as dead code but also the imprecision of binary-level static analysis. A concrete example is when such analysis encounters a system call wrapper like `setxid`: it may mark all the possible system calls that can be made through that wrapper as invoked, independently of those that will actually be made at runtime.

Source-level analysis does not suffer from such issues and as such is more precise than binary-level techniques. For example, on Redis, source-level static analysis reports 71 system calls which is close to the 68 traced at runtime on the test suite. Due to the difficulty of binary-level static analysis, this technique also suffers from a small number of false negatives [2, 10]. Based solely on static analysis, the amount of system calls that would need to be implemented to support all 5 applications is 141 for source-level and 125 for binary-level. We suspect that this smaller number for binary-level analysis may come from its false negatives. We conclude that relying solely on static analysis is not sufficient to get a good understanding of the implementation effort required for an OS aiming at POSIX-like compatibility.

In all, these results bring a message of hope to the level of POSIX compatibilty unikernels can provide: the effort, while non-negligible, is not as insurmountable as past studies relying on static analysis seemed to suggest.

Conclusions

We have argued that the time for wider unikernel deployment is now: the availability of de-facto standard orchestration frameworks such as Kubernetes, coupled with Unikraft's high level of POSIX compatibility, fully modular architecture leading to high performance, and standard but previously sorely missing security features should remove the main barriers to adoption that have in the past crippled unikernel deployment. If you'd like to know more about the project or join the growing Unikraft OSS community please have a look at the project's website at `www.unikraft.org` and please don't hesitate to drop us a line.

References

[1] Per Buer. Unikernels are secure. here is why., 2017. `http://unikernel.org/blog/2017/unikernels-are-secure`, Online, accessed 6/17/2021.

[2] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. Sysfilter: Automated system call filtering for commodity software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*, pages 459–474, 2020.

[3] Antti Kantee and Justin Cormack. Rump kernels no os? no problem! *USENIX; login: magazine*, 2014.

[4] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. Os v - optimizing the operating system for virtual machines. In *Proceedings of the 2014 USENIX Annual Technical Conference*, ATC'14, page 61, 2014.

[5] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ștefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, specialized unikernels the easy way. EuroSys'21, New York, NY, USA, 2021. ACM.

[6] Stefan Lankes, Simon Pickartz, and Jens Breitbart. Hermitcore: a unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS 2016. ACM, 2016.

[7] Anil Madhavapeddy and David J Scott. Unikernels: Rise of the virtual library operating system. *ACM Queue*, 11(11):30, 2013.

[8] Spencer Michaels and Jeff Dileo. Assessing unikernel security, 2019. `https://research.nccgroup.com/wp-content/uploads/2020/07/ncc_group-assessing_unikernel_security.pdf`, Online, accessed 6/17/2021.

[9] musl libc. musl libc - Design Concepts. https://wiki.musl-libc.org/design-concepts.html.

[10] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 59–73, 2019.

[11] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E Porter. A study of modern linux api usage and compatibility: what to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 16. ACM, 2016.