# Unikernels vs Containers: An In-Depth Benchmarking Study in the context of Microservice Applications

Tom Goethals*, Merlijn Sebrechts*, Ankita Atrey*, Bruno Volckaert*, Filip De Turck*

*Ghent University - imec, IDLab, Department of Information Technology
Technologiepark-Zwijnaarde 15, 9052 Gent, Belgium
Email: togoetha.goethals@UGent.be

***Abstract*— Unikernels are a relatively recent way to create and quickly deploy extremely small virtual machines that do not require as much functional and operational software overhead as containers or virtual machines by leaving out unnecessary parts. This paradigm aims to replace bulky virtual machines on one hand, and to open up new classes of hardware for virtualization and networking applications on the other. In recent years, the tool chains used to create unikernels have grown from proof of concept to platforms that can run both new and existing software written in various programming languages. This paper studies the performance (both execution time and memory footprint) of unikernels versus Docker containers in the context of REST services and heavy processing workloads, written in Java, Go, and Python. With the results of the performance evaluations, predictions can be made about which cases could benefit from the use of unikernels over containers.**

***Index Terms*— containers, unikernels, microservices, virtualization, IoT**

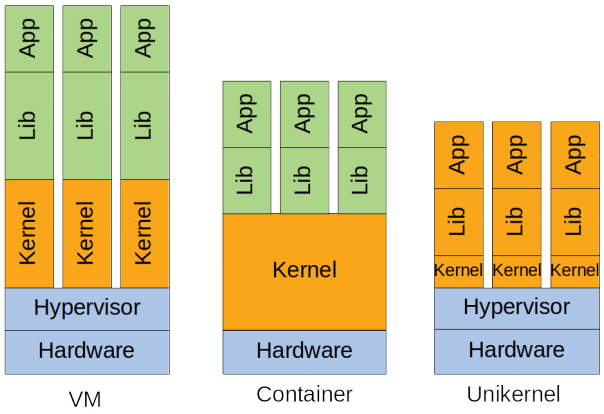


Fig. 1. Comparison of virtual machine, container and unikernel system architecture

## I. INTRODUCTION

Unikernels are a relatively new concept in which software is directly integrated with the kernel it is running on. This happens by compiling source code, along with only the required system calls and drivers, into one executable program using a single address space [1]. Because of this design, unikernels can only run a single process, thus forking does not exist. The build process results in a complete (virtual) machine image of minimal size that only contains and executes what it absolutely needs to. Fig. 1 illustrates this concept by comparing virtual machines, containers and unikernels. The figure uses a blue color to indicate hardware or hypervisors, orange to indicate kernel space and green to indicate user space. The reduced kernel and system complexity can make a unikernel much faster than a regular virtual machine [2]. Despite only being able to run a single process, multi-threading is usually possible [3, 4]. An added advantage of the way unikernels are built is that they are less vulnerable to security problems, since there are typically less attack options, except the program, the required libraries and the kernel functions it uses. The downside is that programs always run in kernel mode, making it easier for bugs and hacks that do succeed to critically break the machine, while making it harder to debug [1] because unikernels usually do not have their own sets of debugging tools and existing ones would have to be cross-compiled. Additionally, all the facilities and libraries used by debugging tools would have to be included, ballooning the size of the unikernel, and any debugging tool that requires another process to run can not work in a unikernel by design. At the time of writing, all platforms generate unikernels as virtual machine images, but work is underway to run them as bare metal images and under Docker [5]. By eliminating the need for a classical operating system, the image size difference with regular virtual machines can be in the order of gigabytes. Additionally, unikernels have a much shorter boot time than full virtual machines, in the order of hundreds of milliseconds compared to tens of seconds, and consume much less memory [6, 7]. The same claims have been made for replacing containers with unikernels, but in this case the advantage in terms of boot time seems to be much smaller [7], except for highly specialized unikernels built around a single language [6].

### A. Applications of unikernels

Many proof-of-concept unikernel versions of all sorts of software exist, including database engines, REST services and a RAMP stack (Rumprun, Apache, MySQL, PHP) [8]. A lot of IoT services are composed of different pieces of software, but these can easily be broken up into a number of individual single process components ready to be converted into a unikernel. Because unikernel images are far smaller than regular virtual machines, a lot of space on cloud infrastructure could be saved by using them. Furthermore,

978-1-7281-0236-8/18/$31.00 ©2018 IEEE
DOI 10.1109/SC2.2018.00008

because unikernels are smaller than virtual machines and boot faster, microservice deployment strategies could use unikernels for more flexible and dynamic deployment [9]. In addition to replacing existing virtual machines and containers, unikernels can also open up new classes of hardware for cloud use that are otherwise unfit to run an entire operating system [9]. This could happen by either running unikernels under a type I hypervisor or by (eventually) running them bare metal. The added advantage of using a hypervisor and unikernels over simply running embedded software (as is now usually the case) is that any unikernel can be easily deployed to any machine in the cloud running a hypervisor and function as a part of a uniform environment. Embedded software on the other hand often requires local access to update or is incompatible with cloud management software.

Broadly speaking, there are two approaches to building unikernels:

- Designed from the ground up around a single programming language, providing a custom (not POSIX compatible) API. At compile time, all the libraries and system calls used by the program are compiled, together with the program itself, creating a single kernel that is started on boot. This type of unikernel platform is generally incompatible with existing source code, requiring a full rewrite using the platform's API. Considering the still-evolving nature of existing platforms of this type, it also often means dropping features that are not yet implementable. On the flip side, this type of unikernel results in superior performance and much smaller images than the other type [2, 11]. IncludeOS and MirageOS are good examples of this approach [16, 17].
- POSIX compatible operating systems that can run existing software by cross-compiling it using existing compilers. These platforms generally have custom kernel implementations and drivers to make them faster, sometimes with options for paravirtualization. Unikernels built this way have larger kernels and resource requirements, but make up for it in ease of use. This type is very useful for converting software that runs on existing virtual machines and containers into unikernels, since the software only needs to be recompiled, not rewritten. Examples include OSv and rumprun [18, 19].

Since the goal of this paper is to compare unikernels and containers as accurately as possible using the same programming languages, the rest of the text will focus on the performance of the second type of unikernels. The reasons for this are that unikernels of the first type, being POSIX incompatible, simply do not have the facilities and API's to run Go, Java or Python.

### *B. Unikernels vs containers*

Since unikernels contain everything from a kernel to user software, they have some important differences from containers:

- A hypervisor is required to run a unikernel, but this also means it can be run on a type I hypervisor, removing the need for a bulky OS.
- A unikernel's built-in kernel, no matter how small, will increase memory use compared to running a program in a container.
- Since a unikernel does not require context switches from user space to kernel space and has simpler device drivers, it should have a significant speed advantage over a container, because a container runs on a much more complex host kernel. Of course, since the kernel of a container can run on bare metal and a unikernel has to be run on a hypervisor, the actual performance will depend greatly on the chosen hypervisor. The results from the tests in this paper indicate that unikernels running on type I hypervisors can in fact give much better performance than containers.
- Unikernels are single process by design, so anything requiring multiple parallel processes must be broken up into separate unikernels. Additionally, breaking software up into separate unikernels will induce at least some communication overhead between the different parts.

## II. RELATED WORK

Many aspects of unikernels have been studied in the past, but due to the changing nature of unikernel platforms, some of the results are likely outdated at the time of writing. Boot time comparisons between virtual machines and unikernels have been made [7], in addition to studies on the boot times of unikernels using a single unikernel platform, often comparing to Linux or Linux virtual machines [6, 11, 10]. Another study used a DNS server and an HTTP server, implemented as unikernels on different platforms, to compare the networking performance of unikernels versus Linux [2]. Despite interesting results, different programming languages had to be used per unikernel platform, so the results depend somewhat on the specific software implementations these languages allowed. An attempt has been made to employ unikernels in an edge offloading architecture [9], allowing for a more dynamic deployment of IoT services, but that work was hampered by a bug that occurred in the chosen unikernel platform at the time. Other studies into this area have been done [11, 12]. The security advantages of unikernels over containers and regular virtual machines have been extensively studied [13, 14, 15]. To the best of our knowledge, a direct comparison of containers versus unikernels using specific software written in the same programming languages has not been done before. Therefore, the rest of this paper will focus only on the performance of software written in Java, Go, and Python, more specifically for microservice applications, in terms of unikernels versus containers.

## III. BENCHMARKING SETUP

This section discusses the potential platforms for the tests, as well as the physical test setup. The tests and measuring methods used to obtain the results are also explained. Several platforms were examined for their ability to create Go, Java and Python unikernels.

### A. Candidate platforms

OSv (0.51) is a unikernel platform that works similar to Docker. At the basis of OSv unikernels lies a very small core to which modules and files can be added through a layered build process. Each layer defines a base image to start from, a number of files to add and optional build and command line commands. Using this method, base images were created for Java 1.8, Python 2.7 and Go 1.10, which were then used to create the unikernels for the tests. Running newer versions of both Java (1.9, 1.10) and Python (3.6) was attempted. While higher Java versions worked, a minimal JRE would not start on OSv due to not being compiled correctly. Java 1.8 was deemed sufficient for the purposes of the tests. In the case of Python 3.6, many system calls were required that are not yet implemented in OSv, so 2.7 had to be used. Modules to run Java 1.8 and Python 2.7 (some modification and building required) are included in the OSv source, while Go code was compiled as a shared library and run indirectly via a wrapper supplied by OSv. To confirm that this wrapper does not have any impact on performance, Go was also compiled as a position independent executable, which can run directly on OSv. A position independent executable is an executable that can execute properly no matter what its absolute address in the address space is [21]. Note that OSv requires all software running on it to be built as position independent by design [22], but in the cases of Java and Python this is already done by cross-compiling the virtual machine and interpreter, respectively. For Go, compiling as a shared library has the same result, except that it still needs to be launched by the aforementioned wrapper. The position independent version of Go is included in the results as Go(pie).

Rumprun (unversioned, Apr 8, 2018 commit) was also considered as a test candidate. Contrary to OSv, rumprun compiles the source code and all required system libraries and drivers in one step, into a single kernel that is loaded during boot. While this allows for more optimizations than OSv's approach, it also results in a slower and bulkier build process. Unikernels for Python and Go were successfully created using rumprun, but during testing these unikernels never managed to complete more than a few thousand requests before running into a socket allocation bug. This problem was partially fixed under KVM (Kernel-based Virtual Machine [20]) by increasing memory, which caused it to happen after a few million requests, but could not be fixed on the test setup. Memory footprint and image sizes of rumprun unikernels were more or less the same as those of OSv unikernels, but the results are not included because they are incomplete and thus unreliable.

UniK [23] (unversioned, Nov 15 2016 release) is a platform that combines several other platforms (including OSv and rumprun) into one tool chain. While this is certainly an interesting development, UniK is less flexible than OSv and rumprun in terms of what libraries and packages are included in any given unikernel (for example, the Java version cannot be changed), and allows less control over how the images are generated and deployed. Despite both rumprun and OSv supporting static IP addresses, no method was found to assign them to unikernels generated with UniK, which was required for the test setup. Additionally, unikernels seem to be (in part) dependent on UniK's daemon to boot, which caused them to hang on XenServer even before running into IP assignment problems. Unikernels for Python and Go were tested on localhost using VirtualBox, but being based on rumprun, they ran into the same problems and thus are not included in the results.

Although several platforms were examined, only OSv was used for testing because it proved to be the most stable option. Rumprun had some quirks that made testing unreliable and UniK, being partially based on rumprun, showed the same symptoms. Additionally, since the more stable parts of UniK rely on OSv anyway, the latter was chosen as a testing platform to reduce the complexity of the build process.

### B. Test machine

All tests were run on an Intel Core i5-2300 processor with virtualization extensions enabled. The machine had a total of 4GB ram and a 160GB Western Digital hard drive. All virtual machines and containers were limited to 256MB RAM, either by configuration or by using Docker's –memory flag. In single threaded tests, both virtual machines and containers were limited to one CPU core and test programs were written to reflect this. For multi threaded tests, all instances were given four cores and the program code was changed to use exactly four threads wherever possible and govern itself where exact numbers could not be forced. The test machine was only used as a server, all client activity was run on a separate machine connected directly to the test machine to avoid result collection from interfering with performance. Only one container or virtual machine was active at any given time during the tests.

Containers were run on Ubuntu 18.04 using Docker 18.03. Container web services were made available by forwarding their ports to the host using docker's -p option (Fig. 3). Unikernels were run on XenServer 7.5, a type I hypervisor [24]. The virtual machines' network interfaces were bridged to the host interface for network access (Fig. 2).

The code for all tests is made available for use and review on GitHub: https://bit.ly/2PTXWZr.

### C. REST service stress test

For Go, Java and Python, a simple REST service was written that contains a static array of to-do items (description, time due and finished/not finished). The service supports the following GET methods:

- /todos: list all items
- /todos/{id}: get the to-do with the specified id

While this is a very simple service, it should be a good indicator of how fast a container or unikernel can process REST HTTP requests and a small amount of code. Apache JMeter [26] was used to run 40 concurrent threads, each firing 50000 requests at the test machine to simulate a good sized concurrent demand. Every request called the /todos method, fetching the entire array as JSON.

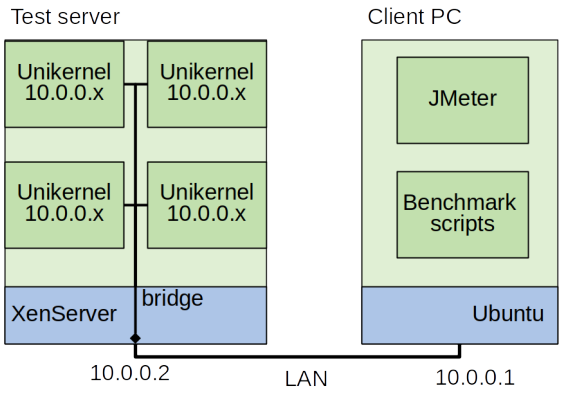


Fig. 2. XenServer test setup for unikernels

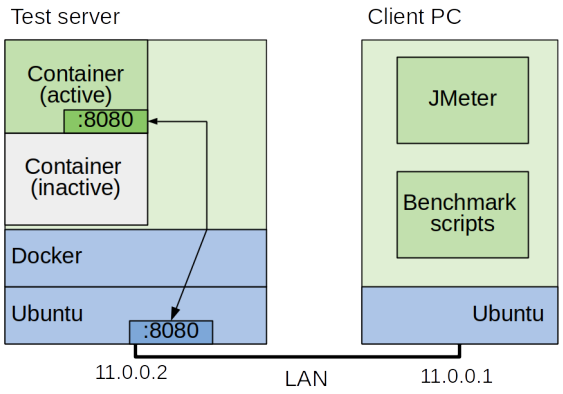


Fig. 3. Docker test setup for containers

The Go web service was created with the standard net/http package in combination with the Gorilla Toolkit mux (v1.6.2) [27]. For the Python version, Flask and Flask-Restful [28] were used. In the case of Java, Vert.X 3.5.1 [29] was used. Vert.x is an event-driven, non-blocking toolkit for developing reactive applications [30] in which a verticle is an atomic piece of deployable code. For this test, a verticle was created that listens to HTTP traffic on a specific port and handles incoming messages like REST service requests.

To enable multi-threading, the verticle was instantiated four times under Java, while Python was given free reign by simply enabling multi-threading for Flask. Go automatically creates a number of threads fitting its hardware environment, so no code changes were necessary [31].

### D. Heavy workload test

For the load test, a simple bubble sort algorithm was implemented as identically as possible for each of the tested programming languages. To focus on processing power and memory load instead of networking overhead, the collection to be sorted was made as big as possible without making testing impractical. The array to be sorted is simply a descending array of numbers (x  0), where x was chosen

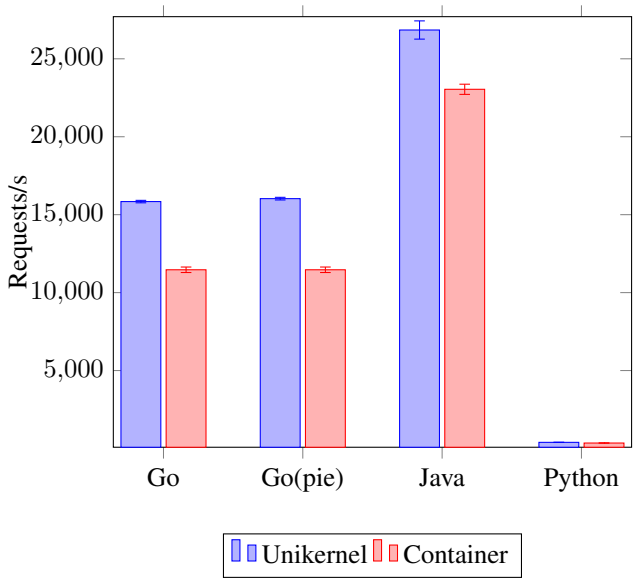


Fig. 4. REST stress test performance evaluation of unikernels versus containers

at 20000. Each test was repeated 20 times to get an accurate average.

This test was built as a web service, so the same frameworks were used as in the REST service stress test. However, these frameworks matter little in terms of performance since the algorithm takes most of the processing time by far.

## IV. RESULTS

In this section the results of the tests are discussed. In addition to raw throughput capacity of the machines, memory footprint and response latency will also be reviewed. Response latency gives an indication of how stable a particular unikernel or container is, while memory consumption is useful in determining if unikernels can be deployed on the same scale as containers.

### A. Single-threaded results

Fig. 4 shows the results for the REST stress test using a single thread to do all processing (higher is better). The Go service is about 38% faster when running as a unikernel than as a container, while the Java version is about 16% faster as a unikernel. The results confirm the claims that unikernels' reduced kernel and complexity makes them a fast alternative to containers, at least when running on a type I hypervisor.

The equality of Go and Go(pie) performance is clearly visible here, confirming that OSvs Go wrapper has no negative effects.

For Python, being an interpreted language rather than a compiled one and thus slower, the result is a bit harder to interpret from the figure. However, the effect from running it in a unikernel seems to be the same, with a performance of $395\pm1$ requests per second for the unikernel versus $351\pm1$ requests per second for the container. This 15% improvement from running in a unikernel is similar to the improvement for Java, but much lower than for Go.

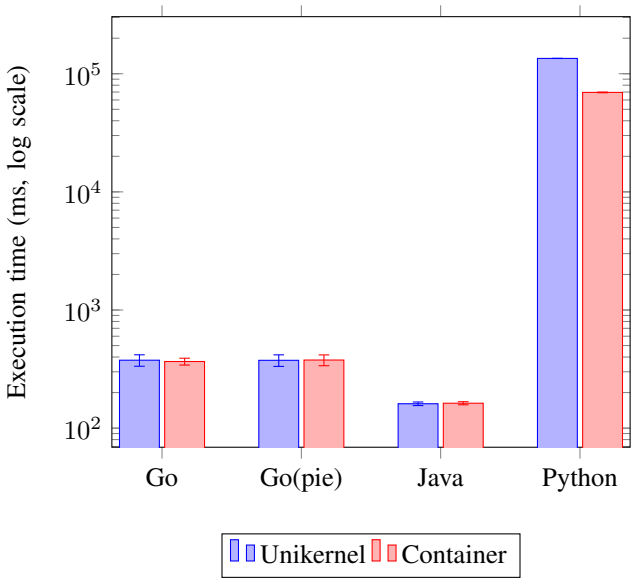


Fig. 5. Bubble sort execution time of unikernels versus containers (log scale, lower is better)

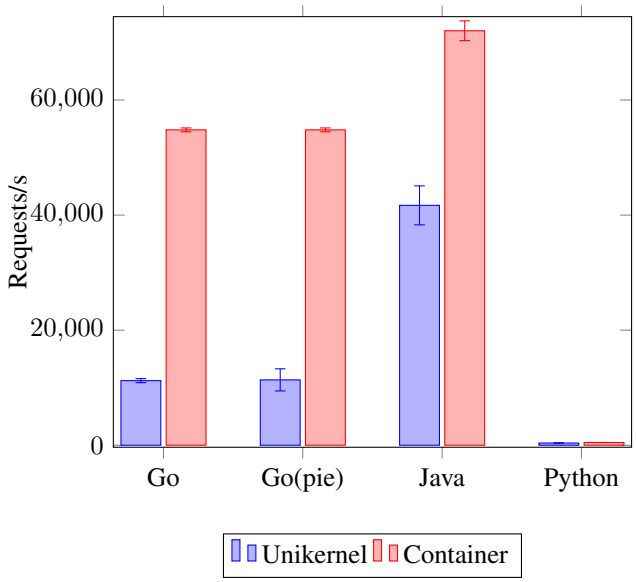


Fig. 6. REST service performance of multi-threaded unikernels versus multi-threaded containers

Fig. 5 contains the results for the heavy workload test (lower is better). Note that a logarithmic scale was used to accommodate Python's performance. The execution times for the Go and Java unikernels seem to be on par with those of their container counterparts. The Go unikernel is about 3% slower (0.5% for the position independent version), while the Java unikernel is about 1% faster. Again, the chart shows that OSv's Go wrapper is doing a good job of avoiding performance penalties.

Python is having some trouble, the execution time of the unikernel is twice as long as that of the container version. This could be because Python uses a disproportionate amount of operations that run slower in a virtual environment. Existing research shows that array and variable access, the building blocks of bubble sort, take up by far the bulk of Python's interpreting overhead [32]. It stands to reason that either or both of these types of instructions run slowly on either XenServer or OSv, and Java and Go may avoid using them so much.

### B. Multi-threaded results

Unikernels can only run a single process, but since they do support multi-threading this aspect merits some attention. Generally speaking, REST services scale almost linearly with the number of cores available to them. Fig. 6 (higher is better) shows good performance scaling for Go and Java containers, but unikernel performance has dropped precipitously. Despite all unikernels performing badly in this case, there are some notable distinctions which are more obvious when compared with the single threaded results.

When directly comparing the multi-threaded and single-threaded results (Fig. 7), only Java shows a performance increase with an increasing number of cores. Even in that case, the only return for quadrupling processing power is 60% more requests per second. Python is relatively unaffected

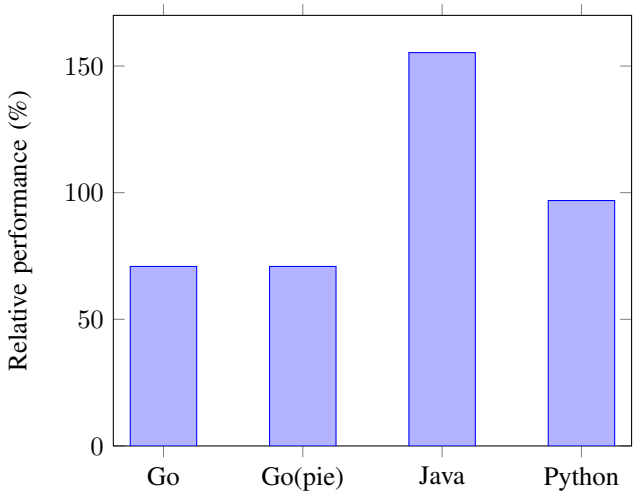


Fig. 7. REST service performance of multi-threaded unikernels relative to single-threaded unikernels

by adding more cores, with multi-threaded performance dipping 3% below single-threaded performance. Go seems to actually suffer a great deal from expanding the thread pool, only managing 75% of its single core performance. It was verified that this is not an effect of Go simply starting too many threads and drowning the scheduler, since Go fetches hardware information and starts as many threads as there are cores [31]. The possibility of XenServer causing these scaling issues was considered, but related work has shown that this is not the case [33], at least in instances where the number of physical cores equals the number of virtual cores and the number of threads is not (much) higher than either, which is true here.

It should be noted that in all cases the combined CPU load of all cores was between 40 and 50% during testing, indicating that a lot of cycles were being wasted on simply

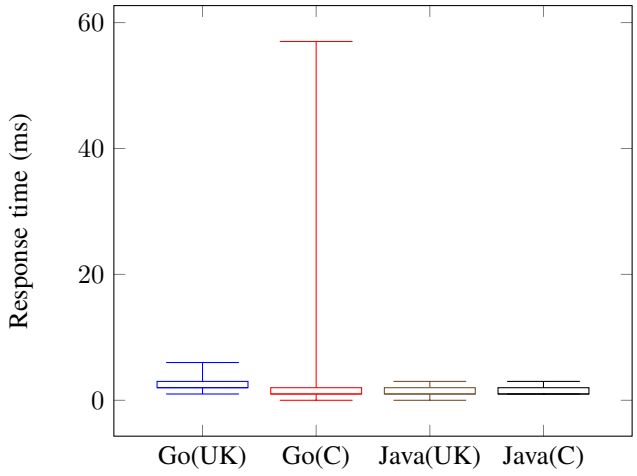


Fig. 8. REST service response time overview for unikernels and containers

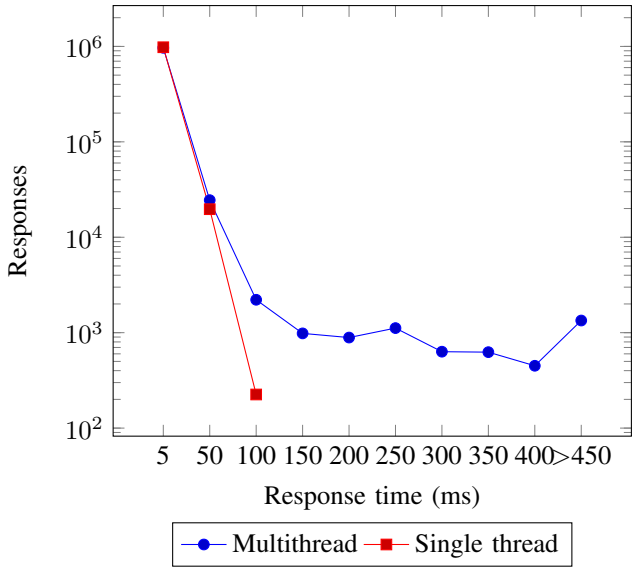


Fig. 9. Number of REST service responses per response time category for single- versus multi-threaded Go unikernels

getting threads to run at all, and not enough on actually running them. This shows that while OSv obviously has very good single threaded performance, any applications using multiple worker threads should probably be split up until threading performance has been stabilized.

### C. Request latency

Part of the requirements for the smooth operation of microservices is having a predictable, stable and preferably low response time. In this section, the results from the stress tests are examined in a different way to see if this is the case for unikernels. Values over the 98th percentile and below the 2nd percentile have been removed to avoid noise from distorting the scope of the charts. All statistics are based on one million requests to their respective web service.

Fig. 8 shows the response times for both Go and Java. A (UK) suffix indicates response times for a unikernel while (C) is the container version. While the Go unikernel seems to have a slightly higher median response time than its container counterpart, the maximum response time of the container version is almost 10 times higher. Taking the performance results from the previous sections into account, its obvious that these numbers are not a problem for unikernel performance. Java, for its part, performs equally well in both cases.

The results for Python were not included in the chart because their range would make the other results hard to interpret. However, the Python unikernel seems to perform considerably better than the container version, with a median response time of 100ms versus 111ms, respectively. Additionally, Python's maximum response times are much better when running as a unikernel with 105ms versus 140ms for the container.

The response times for multi-threaded applications show rather interesting numbers that give some food for thought for the bad scaling performance. Fig. 9 shows the response latencies for both single- and multi-threaded Go unikernels. The response times have been gathered in categories 50ms wide, with the only exceptions being the 0-5ms and >450ms categories. In the single threaded case, the number of responses falls off exponentially with response time, with the maximum response time being 63ms. In the multi-threaded case however, the curve flattens around 100ms and about 0.6% of the requests take a much longer time to complete. 0.12% of the requests even take between 450 and 1000ms to complete. Despite the multi-threaded program actually handling about 99% of all requests slightly faster than the single threaded program, the fact that a small percentage of all requests is held up for a long time makes it slower overall. The only explanation for this is that while the scheduler handles the large majority of threads quickly and correctly, some thread switches are made to wait an exceptionally long time before being able to run their thread and complete their workload.

### D. Memory footprint

Using the REST service images from the stress test, a crude comparison can be made between the memory consumption of unikernels versus containers for the different languages.

Container memory footprint was queried directly from Docker. For unikernels, memory footprint was measured by taking the number of actually allocated pages as reported by VirtualBox for each machine and multiplying it by page size. VirtualBox 5.2 was used for this purpose instead of the already deployed unikernels on XenServer because it was easier to measure memory footprint at any given time using VirtualBox.

Memory footprint was only examined for the single threaded version of unikernels and containers. Measurements were taken after starting an instance and executing a single request to make sure all libraries and variables were initialized. Note that after thousands of requests, memory footprint could be higher than the numbers presented, but

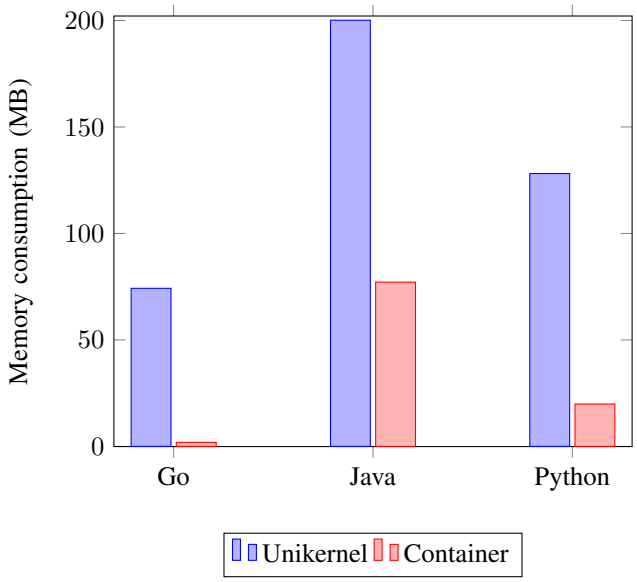


Fig. 10. Container versus unikernel instance memory consumption

would eventually go down again after garbage collection.

Fig. 10 shows that Java unikernels use over twice as much memory than containers, Python unikernels use up to 6 times more memory, and Go unikernels require as much as 30 times more memory. This makes sense, since containers run on top of a host kernel and only need to load their programs into memory. Unikernels, on the other hand, have some memory overhead because of their built-in kernel, no matter how small it may be. The huge difference for Go can be explained by the fact that the program requires no interpreter or virtual machine like Python and Java programs do, so the container version has a minimal memory footprint. This makes the unikernel version, which only requires 70MB more memory in absolute numbers, look comparably huge.

In absolute terms, the extra memory required to run a program as a unikernel is 70MB to 130MB, depending on programming language. One redeeming quality of unikernels is the fact that they can run on a type I hypervisor, eliminating the need for an operating system that could potentially consume a large amount of memory by itself. This means that unikernels are a viable alternative to containers where small to medium numbers of unikernels are concerned, simplistically represented by:

$$HV + UK \times MPK < OS + C \times MPC$$

Where HV is the memory requirement of the hypervisor, OS is the memory requirement of the operating system to be replaced, UK and C represent the number of unikernels and containers respectively and MPK and MPC represent memory requirement per unikernel and container, respectively. Note that the number of unikernels does not always equal the number of containers, since one container may have to be split up into several unikernels due to threading or multi-processing concerns.

### *E. Stability issues*

As discussed before, unikernels under UniK and rumprun had serious stability problems, despite being otherwise fully functional. OSv on the other hand is a stable platform, but there were some quirks:

- When compiling Go as PIE executables, REST services crashed once every 2 to 10 million requests. This was not a huge problem for testing since it was done in batches, but it is something to look out for when planning to use unikernels in any type of production environment. Luckily, the tests have also shown that OSv's wrapper for Go, which is stable, has nearly identical performance, so it would be preferable to run Go that way.
- All OSv unikernels tend to crash every few million requests once they have been multi-core enabled. They did not so much cause errors, but simply hung or stopped without further explanation. Since multi-core performance under OSv proved to be worse than simply instancing several unikernels, this is not much of an issue either.

## V. CONCLUSION

Out of the considered platforms, OSv is the only stable one for the tested scenarios at the moment of writing. It supports (among others) C++, Go, Python and Java, albeit with some small changes to OSv's source code in the case of Python. Other platforms were either unstable during testing and/or were less user friendly, either because it was hard to create usable images in the required format(s) or because the images refused to boot correctly without certain other software in place (UniK daemon). The rest of this section only deals with the results of OSv, so the conclusions may be different for other platforms.

Unikernels exceed containers in terms of pure speed and response time, firmly surpassing them for single core performance on a type I hypervisor. In our evaluations, Java and Python unikernels performed 16% better than their respective containers for a REST service stress test, and Go even performed 38% better. When the focus shifted to heavy workloads, all unikernels kept an equal pace with their container counterparts, apart from the Python unikernel which only reached 50% of the container's performance. Unikernel performance for the REST service stress test can be explained by the fact that this test relied heavily on kernel functions and thus context switches from user space to kernel space. These do not exist in a unikernel, giving unikernels a large advantage over containers in situations where context switches happen very often (REST service), but breaking even in situations where they almost never occur (heavy workload). Another contributing factor is the heavier use of device drivers in the REST service stress test, which are less complex in unikernels.

Unikernels are far from ready for multi-threading, but that should not be a problem for cases where software can either be split up or multiple instances can be deployed

(REST services, modular software) or where multi-threading is not really required (embedded software). In these cases converting software to unikernels could still be a major advantage.

Concerning memory, unikernels consume a good deal more than containers. This makes sense, since unikernels have the extra overhead of a kernel, while containers use the kernel of a host OS. However, since unikernels do not need a full OS to run on in the first place, memory consumption of a hypervisor with a small number of unikernels may be less than that of a large OS running a few containers. The result is that memory consumption is an important factor to consider when deciding whether to use unikernels. Adding more instances will give better performance than adding the same CPU power to one or more containers running the same software, but the trade-off is that the unikernels will use more memory.

In terms of unikernel performance of specific programming languages, Java and Go are the clear winners. Java gives by far the best performance of all, while Go uses the least amount of memory. For new software, this can be useful information when choosing a language for unikernel development. For existing Java or Go software being ported to unikernels, switching to the other language might not be worth the effort required to convert it, unless either speed or memory footprint are absolutely critical.

## VI. ACKNOWLEDGMENT

The research in this paper has been funded by Vlaio by means of the FLEXNET research project.

## REFERENCES

[1] R. Pavlicek, Unikernels, https://www.safaribooksonline.com/library/view/unikernels/9781492042815/
[2] Ian Briggs, Matt Day, Yuankai Guo, Peter Marheine, Eric Eide, A Performance Evaluation of Unikernels, http://media.taricorp.net/performance-evaluation-unikernels.pdf
[3] Sebastian Wicki, The Rumprun Unikernel, pkgsrcCon 2016 https://www.pkgsrc.org/pkgsrcCon/2016/rumprun.pdf
[4] Avi Kivity, OSv - Optimizing the Operating System for Virtual Machines, USENIX ATC'14 Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference Pages 61-72
[5] Mano Marks, Unikernel Systems Joins Docker, https://blog.docker.com/2016/01/unikernel/
[6] Dan Williams, Ricardo Koller, Unikernel Monitors: Extending Minimalism Outside of the Bo x, 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)
[7] Bruno Xavier, Tiago Ferreto, Luis Carlos Jersak, Time provisioning evaluation of KVM, Docker and Unikernels, 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)
[8] M. Lucina, Deploying real-world software today as unikernels on Xen with Rumprun, https://lucina.net/files/rumprun-xpds2015.pdf
[9] Vittorio Cozzolino, Aaron Yi, Ding Jrg Ott, FADES: Fine-Grained Edge Offloading with Unikernels, HotConNet '17 Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems
[10] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, Jon Crowcroft, Unikernels: library operating systems for the cloud, ASPLOS '13 Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems, Pages 461-472
[11] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft and Ian Leslie, Jitsu: Just-In-Time Summoning of Unikernels, Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)
[12] Vittorio Cozzolino, Aaron Yi Ding, Jorg Ott, Dirk Kutscher, Enabling Fine-Grained Edge Offloading for IoT, SIGCOMM Posters and Demos '17 Proceedings of the SIGCOMM Posters and Demos, Pages 124-126
[13] Bob Duncan, Alfred Bratterud, Andreas Happe, Enhancing cloud security and privacy: Time for a new approach?, 2016 Sixth International Conference on Innovative Computing Technology (INTECH)
[14] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, Felipe Huici, My VM is Lighter (and Safer) than your Container, SOSP '17 Proceedings of the 26th Symposium on Operating Systems Principles Pages 218-233
[15] Alfred Bratterud, Andreas Happe, Robert Anderson Keith Duncan, Enhancing Cloud Security and Privacy: The Unikernel Solution, Eighth International Conference on Cloud Computing, GRIDs, and Virtualization, 19 February 2017 - 23 February 2017, Athens, Greece
[16] IncludeOS, includeOS, http://www.includeos.org/
[17] Xen and Linux Foundation, MirageOS, https://mirage.io/
[18] Cloudius Systems, OSv, https://github.com/cloudius-systems/osv
[19] Rumprun, https://github.com/rumpkernel/rumprun
[20] Kernel-based Virtual Machine, https://www.linux-kvm.org/page/Main_Page
[21] Introduction to the Application Loader, ftp://bitsavers.informatik.uni-stuttgart.de/pdf/intel/iRMX/iRMX_86_Rev_6_Mar_1984/146196_Burst/iRMX_86_Application_Loader_Reference_Manual.pdf 1-3
[22] Nadav Har'El, Running compiled code on OSv, https://github.com/cloudius-systems/osv/wiki/Running-compiled-code-on-OSv
[23] Solo.io, UniK, https://github.com/solo-io/unik
[24] XenServer - open source virtualization, https://xenserver.org/
[25] Unikernel test code repository, https://github.com/togoetha/unikernels-v-containers
[26] Apache JMeter, https://jmeter.apache.org/
[27] Gorilla web toolkit: mux, http://www.gorillatoolkit.org/pkg/mux
[28] Flask RESTful, https://flask-restful.readthedocs.io/en/latest/
[29] Eclipse Vert.x, https://vertx.io/
[30] Hseyin Akdoan, An introduction to Vert.x, https://dzone.com/articles/introduce-to-eclicpse-vertx
[31] Go 1.5 Release Notes, https://golang.org/doc/go1.5
[32] Gerg Barany, Python Interpreter Performance Deconstructed, Dyla'14 Proceedings of the Workshop on Dynamic Languages and Applications Pages 1-9
[33] Cong Xu, Yuebin Bai, Cheng Luo, Performance Evaluation of Parallel Programming in Virtual Machine Environment, 2009 Sixth IFIP International Conference on Network and Parallel Computing