Benchmarking Parallelism in Unikernels

Akilan Selvacoumar$^{1[0000-0002-9837-3432]}$, Robert Stewart$^{1[0000-0003-0365-693X]}$, Hans-Wolfgang Loidl$^{1[0000-0001-6318-1732]}$, and Ahmad Ryad Soobhany^{1}

Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, UK as251@hw.ac.uk

Abstract. Virtualisation technologies are widely used in Cloud computing infrastructures, because they can be provisioned cheaply and quickly to meet demand. The common approaches are either to package a Operating System (OS) as a Virtual Machine, or to containerise software with an OS kernel. An emerging alternative are unikernels, which are customised kernels to support just one application. Unikernels are lightweight and an applications has sole use of the kernel, which offers potential for fast, resource efficient and secure execution. For these reasons, unikernels may be idea for parallel computing in the Cloud. However, the parallel performance of unikernel-based Cloud applications has not been extensively studied. This paper presents an evaluation of the OSv unikernel using a parallelised Mandelbrot benchmark, comparing with Docker and a monolithic VM for runtime, parallel speedups and boot-up time. OSv has the fastest boot-up time, and is comparable with the parallel speedups of Docker and the monolithic VM.

Keywords: Unikernels · Parallel Computing.

1 Introduction

1.1 Unikernels for the Cloud

The convention for running parallel programs is by executing them on multicore CPUs within standard Operating Systems (OS) with multithreading support. Security, on-demand elastic scalability and energy efficiency are the primary requirements when deploying parallel programs to the Cloud [19–21], which is why virtualisation technologies are widely used for Cloud deployments. The most widely used virtualisation technologies are application-specific containers e.g. Docker, and OS Virtual Machines e.g. VirtualBox and Qemu. These approaches package the full OS software stack along with compiled applications.

An emerging alternative approach is unikernels [2]. A unikernel is an executable image that can execute natively on a hypervisor, without the need for a separate operating system. A unikernel are application specific, i.e. they are customised to execute a single binary program. A unikernel is a lightweight OS kernel where the kernel modules are shared with the hypervisor e.g. Qemu. This results in quicker boot times because there are fewer modules to start, and additional security guarantees are provided because the compiled application is the only entry point.

Unikernels are designed to be fast, customisable and secure. Unikernels have a single address space, for the one application being executed. This minimises context switching only to kernel actions rather than to other OS processes. Unikernels have a single address space, for the one application being executed. This minimises context switching only to kernel actions rather than to other OS processes.

All components are modularised including operating system primitives, drivers, platform code and libraries should be easy to add and remove as needed, to generate a light weight and flexible operating system. This helps in terms of reducing overhead for the OS and provides the flexibility to switch between different OS modules. POSIX support provides the ability to run existing legacy applications. exposure to external security threads is minimised because no other process is executed in the virtualised environment.

There are two distinct approaches to unikernel implementation. The first approach is to develop unikernels for specific programming languages, where the low-level kernel libraries are developed in one language and applications are developed in the same language. Examples include MirageOS [4] for OCaml programs and HaLVM [3] for Haskell programs. Any static security guarantees of the language, e.g. through its type system, are guaranteed for the complete software stack. The second approach is to develop unikernels that are compatible with standard binaries. That is, any programming language compiled to native code are supported by these kinds of unikernels. Examples include OSv [5] and Unikraft [6].

The experiments in Section 3 uses the language agnostic OSv unikernel. This is because it supports multiple programming languages with parallelism features, and is portable across both hypervisors (e.g. Qemu and Xen) and also CPU architectures (e.g. ARM and x86).

1.2 Parallel Performance of Unikernels

Existing unikernel benchmarks have focused on latency and boot-up time [4] and CPU performance [8]. Another metric is OS noise, which is benchmarked in [7] using FWQ (Fixed work Quanta), FTQ (Fixed time Quanta) and hourglass metrics. When compared with the Linux kernel, the Azalea Unikernel [7] had less kernel interference and scaled well to many-core CPUs. Other benchmarks focus on specific application domains, e.g. comparing boot-up times, file read/write latencies and memory allocation for the Sqlite and Redis databases [6].

The lightweight nature of unikernels, and the fact that programs have exclusive use of the kerne, may enable them to achieve good parallelism efficiency. To the best of the author's knowledge, there is little work on profiling the parallel performance of unikernels. This paper provides a systematic evaluation of a parallelised Mandelbrot implementation on three virtualised software stacks: Docker, a virtualised Linux OS and the OSv unikernel. Section 3 compares boot-up times, wall-clock runtimes and parallel speedups.

The aim of this paper's experiments are to provide insights into the comparative performance of parallel computing with unikernels. The wider context

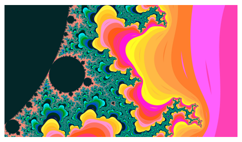


Fig. 1. A generated Mandelbrot image

of this PhD is to firstly discover parallel performance bottlenecks of emerging unikernels, to identify systems-level research opportunities and to find ideal use cases for parallel computing with unikernels.

2 Experimental Design

2.1 Benchmark metrics

Wall clock run times The wall clock runtime measures in seconds the total time to execute the Mandelbrot [16] program. The wall clock runtime does not include the boot-up time. This metric shows the sequential runtime performance, as well as the parallel runtime performance on multiple CPU cores. The standard deviation, parallel speeds and parallel efficiency are calculated based on the wall clock runtimes.

Boot-up times The boot-up times is the time taken to the start the Virtual Machine or Container as well as starting the application's execution. This metrics shows how fast it takes to start a parallel program, e.g. if deployed on-demand in the Cloud.

Parallel speedups The parallel speed up is defined as the ratio of serial execution time to the parallel execution time [17]. This metric shows how well Mandelbrot speeds up with multiple CPU cores versus the sequential runtime with the same virtualised software stack, e.g. OSv on 8 cores and OSv on 1 core.

2.2 The Mandelbrot Benchmark

Mandelbrot images are generated by applying a mathematical function to each complex number projected in the complex plane and determining for each whether they are bounded or escapes towards infinity. The experiments in Section 3 are ran with two sets of Mandelbrot parameters, image height and iterations.

These values are a height of 1000 with 3000 iterations, and then a height of 2000 with 6000 iterations. This is to evaluate the parallel performance of three virtualisation comparators when computational complexity increases.

The Mandelbrot algorithm is parallelised using Goroutines in Go. A Goroutine is spawned for each row to be generated in the Mandelbrot image. Inside each parallel Goroutine is a sequential loop which iterates through each image column. The loop executes the Mandelbrot iteration and Linear Interpolation. Once all go routines are executed the image is written from to disk (Figure 1).

2.3 Comparators

Existing unikernel benchmarks [6, 8] use Docker and Monolithic kernels as their standard means of comparison. Docker is commonly used for Cloud deployment given its ability to rapidly spawn containers for elastic-scale computing.

There are three virtualisation comparators in the experiments (Section 3):

1. OSv running on a Qemu emulator. The Mandelbrot program (implemented in Go) is compiled using Cgo, and the generated shared object and header files are linked to the OSv kernel and then executed.
2. Docker running a Ubuntu 20.04 image. The Mandelbrot program is compiled with the Go compiler which is linked to the Docker file then executed.
3. A monolithic kernel (Ubuntu 20.04) running on a Qemu emulator. The Mandelbrot program is also compiled with the Go compiler, then executed in userspace as a binary file.

Both Qemu and Docker are ran on a host OS (i.e. type-2 hypervisor) in the experiments. The hardware specification of the machine is a Intel i7-1065G7 CPU with 8 cores and 16 GB of memory. For the experiments in Section 3, the type 2 hypervisors ensure a fair comparison between OSv, the monolithic kernel and Docker at the same layer of virtualized abstraction.

3 Results

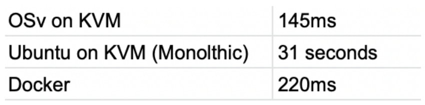
3.1 Wall Clock Run times

The wall clock runtime results for both parameters are in Figures 3(a) and 3(b). The plots of the wall clock runtimes refer to the mean of run times. The x-axis refers to the number of cores used and the y-axis refers to seconds of the mean run time all measurements are the mean of 3 executions.

Each experiment is executed 8 times to obtain the mean average and the standard deviation.

Scenario 1: The result in scenario 1 in Figure 3(a) shows that OSv is slower than Docker and the Monolithic kernel (i.e. Ubuntu). The Monolithic kernel and Docker have almost identical run times with a difference of 2 seconds (circa 1% of the single core runtime). OSv on the other-hand exhibits runtimes that are up to 8.5 seconds higher than Monolithic kernel and Docker. For Scenario 1 (Figure 3(a)), OSv is the fastest system across all core numbers, 32 seconds faster than the Monolithic kernel and 29 seconds faster than Docker on 4 cores. On 6 and 8 cores OSv runs on average 19 seconds.

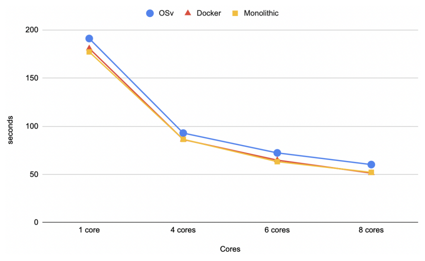
Across all parameters and core numbers, OSv was slower than the other 2 comparators by a difference on average of 17 seconds. With higher core numbers



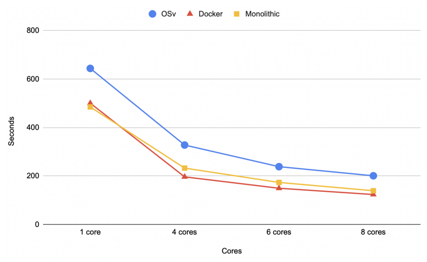
OSv on KVM	145ms
Ubuntu on KVM (Monolthic)	31 seconds
Docker	220ms

(a) Bootup Times

Fig. 2. Bootup times

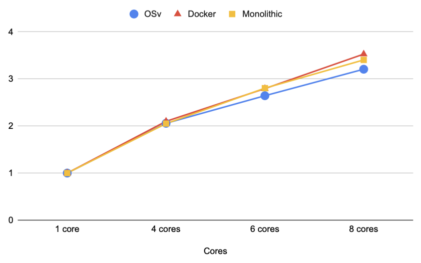


(a) Wall Clock Scenario1

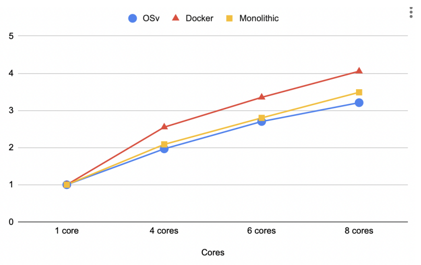


(b) Wall Clock Scenario2

Fig. 3. Wall clock run times



(a) Parallel Speed ups Scenario 1



(b) Parallel Speed ups Scenario 2

Fig. 4. Parallel Speed ups

the differences in runtimes between the systems (both in absolute time and percentages) decreases. In Scenario 1 with one core OSv runs faster than Docker by 24 seconds and OSv is slower than the Monolithic Kernel (i.e. Ubuntu) by 3 seconds. In all other runs of Scenario 1 OSv is *slower* than the other 2 comparators by 20.5 seconds. In the case of OSv for Scenario 1 the first run of each core number delivered the fastest run time.

Scenario 2: In Scenario 2 in Figure 3(b) OSv is slower than Docker and the Monolithic kernel. Docker consistently has the fastest mean wall clock run times compared to the other two systems: it is on average 23 seconds faster than the Monolithic kernel, and 102 seconds faster than OSv. In Scenario 2 (Figure 3(b)), and in contrast to Scenario 1, OSv is consistently *slower* than the other configurations: across all core numbers it runs by 82 seconds slower than Docker, and by 99 seconds slower than the Monolithic kernel. In a single core configuration, we observed the highest differences in runtime for OSv, by an average of 113 seconds slower than Docker and 130 seconds slower than the monolithic kernel. On a single core configuration OSv had the least stable results with a standard deviation of 57.84 seconds. In comparison Docker had a standard deviation of only 40 seconds and the Monolithic kernel had the lowest standard deviation of 16.4 seconds. Interestingly, for the multi cores runs OSv had more stable run times in comparison to Docker and the Monolithic kernel, with standard deviations of 0.4 for OSv, 1.45 for Monolithic kernel, and 9.61 for Docker. This means although Docker and Monolithic kernel had faster run times in the multi-core configurations, OSv delivers more predictable run times based on the standard deviation calculated across the 8 runs on different core numbers.

3.2 Boot up times

Figure 2(a) shows the average boot times of OSv, Docker and monolithic kernel (i.e Ubuntu). In terms of boot-up times OSv has faster than Docker by 25% and 99% faster than the monolithic Kernel (i.e Ubuntu).

3.3 Parallel Speedups

The parallel speed ups are in Figure 4(a) and Figure 4(b). They show the parallel speed ups of all configurations, calculated based on the mean wall clock run times over 3 runs.

Scenario 1: The parallel speed ups for OSv multi core is respectively 2.05x times for 4 cores, 2.64x for 6 cores and 3.20x 8 cores. OSv achieves modest speedups of 2.05× on 4 cores, 2.64× on 6 cores, and 3.20× on 8 cores In comparison, Docker achieves initially higher but relatively dropping speedups of 2.10× on 4 cores, 2.79× on 6 cores. and 2.80× on 8 cores. Finally, the Monolithic kernel achieves similar speedups with the best high-end performance of 2.05× on 4 cores, 2.80× on 6 cores,and 3.40× on 8 cores. Based on these parallel speed ups for OSv the parallel efficiency is 51% on 4 cores, 45% on 6 cores, and 39% on 8 cores. The parallel efficiency for Docker is 52% on 4 cores, 46% on 6 cores, and 44% on 8 cores. The parallel efficiency for the Monolithic kernel is 51% for 4 cores, 46% for 6 cores and 43% for 8 cores. For this scenario Docker and the Monolithic kernel have a similar speed up compared to OSv with an average difference of 20%. Docker and the Monolithic kernel have a similar parallel efficiency compared to OSv with an average of 2% parallel efficiency difference.

Scenario 2 On Senario 2, OSv achieves modest speedups of 1.96× on 4 cores, 2.70× on 6 cores, and 3.20× on 8 cores. In comparison, Docker achieves higher speedups of 2.54× on 4 cores, 3.35× on 6 cores, and 4× on 8 cores. Finally, the Monolithic kernel achieves speedups of 2.08× on 4 cores, 2.80× on 6 cores, and 3.49× on 8 cores.

3.4 Discussion

The boot-up time comparison shows that launching applications with OSv is slightly faster than doing so with Docker and significantly faster than with a monolithic VM. This is likely because Unikernels have fewer modules to launch compared to the other two. The more reproducible run times with Scenario 2 for OSv likely because there are fewer background processes and no other user-level applications running which results in less OS noise. The slightly longer runtimes for OSv may be due to the OS multi-threading scheduling implementation in OSv. As future work, will plan on investigating this.

4 Conclusion

This paper presents an experiment comparing the parallel and boot-up time performance of the OSv unikernal compared with a Docker container and a monolithic Linux VM. The results (Section 3) provides a better understanding on how Unikernels preform on parallel applications, and what needs deeper investigation. The unikernel achived the fastest boot-up time. Moreover these experiments show that the OSv unikernel can achieve parallel speedups. Moreover these speedups are comparable with Docker and the Linux VM. This provides a starting point for deeper investigation in this PhD research. The future work will focus on benchmarking parallelised applications from a wider range of domains, using more metrics including memory profiling, energy consumption and OS noise.

References

1. Wu, Song, et al. "Android Unikernel: Gearing Mobile Code Offloading towards Edge Computing." Future Generation Computer Systems, vol. 86, Sept. 2018, pp. 694–703. ScienceDirect, https://doi.org/10.1016/j.future.2018.04.069
2. Madhavapeddy, Anil, et al. "Unikernels: Library Operating Systems for the Cloud." ACM SIGARCH Computer Architecture News, vol. 41, no. 1, Mar. 2013, pp. 461–72. March 2013, https://doi.org/10.1145/2490301.2451167.

3. Cheon, J., Kim, Y., Hur, T., Byun, S. & Woo, G. An Analysis of Haskell Parallel Programming Model in the HaLVM. *Journal Of Physics: Conference Series.* **1566**, 012070 (2020,6), https://doi.org/10.1088/1742-6596/1566/1/012070
4. Madhavapeddy, A. & Scott, D. Unikernels: Rise of the Virtual Library Operating System: What If All the Software Layers in a Virtual Appliance Were Compiled within the Same Safe, High-Level Language Framework?. *Queue.* **11**, 30-44 (2013,12), https://doi.org/10.1145/2557963.2566628
5. Kivity, A., Laor, D., Costa, G., Enberg, P., Har'El, N., Marti, D. & Zolotarov, V. OSv—Optimizing the Operating System for Virtual Machines. *2014 USENIX Annual Technical Conference (USENIX ATC 14).* pp. 61-72 (2014,6), https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity
6. Kuenzer S, Bădoiu V-A, Lefeuvre H, Santhanam S, Jung A, Gain G, Soldani C, Lupu C, Teodorescu Ş, Răducanu C, Banu C, Mathy L, Deaconescu R, Raiciu C, Huici F (2021) Unikraft: fast, specialised Unikernels the easy way. In: Proceedings of the Sixteenth European Conference on Computer Systems. ACM, Online Event United Kingdom, pp 376–394
7. Cha, S., Jeon, S., Jeong, Y., Kim, J. & Jung, S. OS noise Analysis on Azalea-Unikernel. *2022 24th International Conference On Advanced Communication Technology (ICACT).* pp. 81-84 (2022)
8. Xavier, B., Ferreto, T. & Jersak, L. Time Provisioning Evaluation of KVM, Docker and Unikernels in a Cloud Platform. *2016 16th IEEE/ACM International Symposium On Cluster, Cloud And Grid Computing (CCGrid).* pp. 277-280 (2016)
9. What Is Rally? — Rally 3.3.1 dev7 Documentation. https://rally.readthedocs.io/en/latest/. Accessed 19 May 2022.
10. OSProfiler, 05 2022, [online] Available: https://github.com/stackforge/osprofiler.
11. Projects — Unikernels. http://Unikernel.org/projects/. Accessed 14 May 2022.
12. Tajbakhsh, M. & Bagherzadeh, J. Microblogging Hash Tag Recommendation System Based on Semantic TF-IDF: Twitter Use Case. *2016 IEEE 4th International Conference On Future Internet Of Things And Cloud Workshops (FiCloudW).* pp. 252-257 (2016)
13. Boettiger, C. An Introduction to Docker for Reproducible Research. *SIGOPS Oper. Syst. Rev..* **49**, 71-79 (2015,1), https://doi.org/10.1145/2723872.2723882
14. Prabhakar, R. & Kumar, R. Concurrent Programming in Go.
15. "Why Gleam · Chrislusf/Gleam Wiki." GitHub, https://github.com/chrislusf/gleam. Accessed 26 May 2022.
16. Selvacoumar, A. AKILAN1999/Mandelbrot-go-uni-kernel-. *GitHub.*, https://github.com/Akilan1999/mandelbrot-go-uni-kernel-
17. El-Nashar, A. ArXiv.org e-print archive. *TO PARALLELIZE OR NOT TO PARALLELIZE, SPEED UP ISSUE.*, https://arxiv.org/pdf/1103.5616.pdf
18. Makkad, S. Mandelbrot Set Basics. *Fractal To Desktop.* (2018,12), https://fractaltodesktop.com/mandelbrot-set-basics/index.html
19. Bratterud, A. Enhancing cloud security and privacy: The unikernel solution. , https://aura.abdn.ac.uk/bitstream/handle/2164/8524/AAB02.pdf
20. Bratterud, A. A framework for elastic execution of existing MPI programs. *IEEE Xplore.*, https://ieeexplore.ieee.org/abstract/document/6008941
21. Fontana de Nardin, I., Da Rosa Righi, R., Lima Lopes, T., André da Costa, C., Yeom, H. & Köstler, H. On revisiting energy and performance in microservices applications: A cloud elasticity-driven approach. *Parallel Computing.* **108** pp. 102858 (2021), https://www.sciencedirect.com/science/article/pii/S0167819121001010