Research Analysis - Unikernels

Cristian Vijelie, Cătălin Pușcoci, Cezar Crăciunoiu, Dragoș Argint
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
Splaiul Independenței 313, Bucharest, Romania, 060042
dragosargint21@gmail.com *cezar.craciunoiu@gmail.com*
cristianvijelie@gmail.com *catalin1999puscoci@gmail.com*

January 21, 2022

1 Introduction

There are a lot of different unikernel projects, all oriented on different sought-after aspects. Some favor performance, others favor security. One thing all unikernel projects offer is high specialization. This is done to ensure that only the minimal set of features is in the image. By offering modularization, the unikernel user will be tasked with finding the best combination of libraries that suits their usage and their sought-after aspects mentioned above. Overall, these unikernels have one thing in common, reducing the code base, resulting in smaller images, and smaller trusted control bases.

All papers presented below touch on the unikernel subject and show cast it as a means to an end. The unikernels presented are used to host different applications and are compared with the fully-fledged kernel alternatives like Linux. Unfortunately, there isn't one single application tested by all 3 papers, and, as such, it is harder to compare them.

2 *IncludeOS: A minimal, resource efficient unikernel for cloud services*

This paper[1] focuses on a single single, implemented, unikernel, named *IncludeOS*. The writers focus on their development process and the problems that they encountered, and their results compared to full OSes like Ubuntu, but also make some comparisons to MirageOS (another unikernel) on a theoretical level. Their task was to bring resource efficiency to an extreme, which they somewhat succeeded.

IncludeOS is one of the first "modern" unikernels that appeared since their recent reemergence. Although support for it seems to have dwindled in the last couple of years, it is still a viable alternative. The paper presents ambitious features for an operating system, like no interrupts support and fully asynchronous execution. This was novel at the time when the paper was written (2015), and certainly offered another perspective on OS programming.

2.1 Strengths

IncludeOS offers many novel ideas that change the way one would program an OS. These can be seen as advantages, but can also hinder the development speed, as developers need to get acquainted with the changes.

2.1.1 C++ language development

Programming operating systems in C++ can significantly increase the development speed of the operating system. Moreover, by using this C++ environment, developers can have much more abstractization available which comes in handy for developing operating systems. Performance isn't a problem either, C++ programs being almost as fast as the C ones. Of course, there is also the advantage of compatibility with C, which means that most of the existing operating systems can become an inspiration source.

This comes at a cost though. By using C++, working with low level operations becomes more unintuitive. Moreover, the image created through compiling C++ code is harder to inspect and is usually a drawback. Finally, the C++ standard is a bit more volatile than C, but this should not impact the development.

Summing all things up, developing OSes in C++ comes with many more advantages than disadvantages. Using C++ keeps the advantages of C, whilst offering a more structured way of arranging your code and, as such, reducing accidental bugs. The writers of the paper highlight all of this, and how they took advantage of even more C++ features like STL, for example.

2.1.2 Fully asynchronous execution & No interrupts

This is an unusual goal for an operating system, but, an ingenious one. Many operating systems have trouble implementing asynchronous support, and, when done, it usually comes at a cost and most developers are taken by surprise by it. By having only asynchronous execution, most OS components implementations that would have to support it, like the scheduler, become greatly simplified. This goes hand in hand with their goal of greatly reducing the image size. This means that most of the scheduling functionality is kept, with fewer lines of code.

Moreover, by supporting only asynchronous execution, developers have a clear mindset and can't make confusions (much like JavaScript), again reducing the chance of bugs.

2.1.3 Usual, easy to understand use case

The use case presented, a simple DNS server, is not too complex and can be understood by any reader. This came as an advantage to the writers, as this use case best suited their available components. As they did not have TCP support yet, they were limited to UDP services. Moreover, this minimal DNS server does not require much work to be done to offer all its basic features. All in all, it was cleverly chosen to highlight most of the strengths of IncludeOS, which is a good thing for the paper and the writers.

2.2 Weaknesses

At the state presented in the paper, the unikernel presented more drawbacks than advantages. It was presented more like a proof of concept, instead of the beginning of a potentially big project.

2.2.1 No virtual memory

The paper presents the absence of virtual memory as a big upside of IncludeOS. From the aspect of size, they are right. By not including virtual memory, they reduce much of the code of the operating system, but, as they mention, this does not come with a tangible performance increase as first thought. Sometimes IncludeOS is faster, sometimes it is slower.

They stay true to the saying of "extreme specialization", but do not take into account that, in this case, the plus of performance and size outweighs the minus of security. Moreover, by having to deal with segments, one may argue that the code is more bug-prone, as this was generally a headache for programmers. All in all, they lose more than they gain by not including it, but at least they focus on their main goal: the binary size.

2.2.2 Unfair performance comparison

This is one of the biggest drawbacks of the paper. They give examples of other minimal operating systems and unikernels (like MirageOS) in the beginning, but completely ignore them when doing tests. Most test compare 4 cases: IncludeOS VM, Ubuntu32/64 VMs, and Ubuntu Native. Of course a minimal operating system will fare better in size and performance than a usual, fully-fledged, operating system. Even like this, the results they get is that IncludeOS is faster, but not by much.

From the writers' perspective, they did not choose tests that would truly highlight the advantages of IncludeOS and how it fares to the alternatives (MirageOS). The general feel of the benchmarks is that they went with general comparisons that only highlight that IncludeOS barely holds up, even though it is sometimes faster, whilst being hundreds of times smaller than a Linux distribution.

All in all, the writers seem unsure with what they wanted to achieve through the benchmarks, but it's good that they present both upsides and downsides.

2.2.3 X86-only support

Focusing on a single architecture reduces development, but also reduces the exposure the unikernel receives. It is understandable, why they did it. X86 is easier to implement and covers many of the usual use cases. At the same time, it might allow them an easier removal of virtual memory. Still, devices using the ARM or AVR architecture might be better candidates for running IncludeOS bare-metal.

2.2.4 Minimal proof of concept

Even though in 2020 IncludeOS is much more feature-complete, at the moment when the paper was written (2015), the writers worked on bringing IncludeOS into a functional state and then shifted their attention to results. This can be seen in the "Future Work" section.

They write about general future ideas, but no concrete next steps that they should take to upgrade IncludeOS. Finally, they test one of the simplest cases implementation-wise, a DNS server. This requires little OS-related implementation and will always be one of the first targets for new projects. This does not mean, though, that it should be part of a paper proof of concept.

2.3 Alternative approaches

The writers take some interesting alternative approaches to the whole idea of operating systems, two of the most impactful being described below. These features are not objectively good or bad, but offer an alternative to the usual minimal operating systems.

2.3.1 No paging/virtual memory

Removing virtual memory is one of the rarer choices you would see. The writers feel confident that this feature offers a general performance increase, and they accept the difficulty that comes when programming without virtual memory. As this is usually accepted as standard now, not many developers would want to work without virtual memory day to day. This performance increase comes with many drawbacks.

2.3.2 Asynchronous execution

This might come as a surprise to most people. Asynchronous execution is usually no preferred by inexperienced people, but combining it with synchronous execution is certainly worse. By switching to a fully asynchronous execution, the writers made things clear from the start and also simplified much of the design of parts of the operating system

3 *Unikernels: Library Operating Systems for the Cloud*

In this article [2], the authors introduce MirageOS, a Unikernel written in a high level language OCaml [3]. As stated before, Unikernels can be seen as small and highly specialized Virtual Machines, i.e. single-purpose VMs focusing on the application. Unikernels are based on classic library Operating Systems, where each component, such as memory management, IO, and scheduler, is modular and may be added as needed. The downside of these operating systems is that hardware compatibility is not guaranteed. Unikernels keep the flexibility of the library OSs, but they strive to make use of the virtualization support offered by modern CPUs and hypervisors. Due to the small size of the images, Unikernels bring substantial benefits to Cloud architectures where resources can be used more efficiently. Also, small images mean a reduced attack surface. Moreover, running above the hypervisor, which is part of the TCB (Trusted Computing Base), provides isolation and therefore brings further security benefits.

3.1 Strengths

Being one of the pioneers of Unikernels (the year the article was published is 2013), the authors bring innovation in the field of operating systems. The article may be used as a reference for anyone who wishes to learn more about a Unikernel, including what it is, what benefits it provides, and what limits it has.

3.1.1 Sealing the VM and ASLR

Unikernels are single address space (SASOS) operating systems that use minimalist software components and are statically compiled into a few MB image. Based on these principles, the authors introduce the idea of sealing the VM to reduce code injection attacks. To do this, the Unikernel must utilize a page table in which the page cannot be both writable and executable, and then execute a hypercall that prohibits the page table from being further changed. On the other hand, the VM sealing feature would leave the Unikernel vulnerable to ROP attacks. Because these types of attacks are usually mitigated using ASLR, and it requires a runtime linker, which would introduce complexity into the Unikernel, the authors come up with another clever solution: ASLR at compile time. Due to the way Unikernels are deployed, similar to containers, it is easy and fast to reconfigure them and therefore to introduce ASLR in the compilation process.

3.1.2 Performance and Use Cases

The article places a lot of value on performance appraisal. They analyzed various conventional scenarios such as networking, block IO, thread generation. By default, Unikernels employ zero-copy, reducing the cost generated by traditional OSes.

One of MirageOS use cases is OpenFlow Controller, a software-defined networking standard for Ethernet switches. It's interesting to see how they were able to match the performance of Maestro and NOX, the other two implementations on the market at the time.

Another important aspect is that, unlike classic VMs, Unikernels boot very fast - "Unikernels are compact enough to start and respond to real-time network traffic". This is important because we can use the resources of a cloud infrastructure more efficiently by pausing the VM when it is not needed and resuming it when it is.

3.2 Weaknesses

Since they are a relatively new method of virtualization, Unikernels also have many disadvantages. Some disadvantages are highlighted in the article, others are easily deductible.

3.2.1 OCaml

Although there are conventional systems languages such as C, the authors opted to create MirageOS in OCaml, a high-level programming language, that provides better security and type safety. The fact that MirageOS is written almost exclusively in OCaml is a weakness because it has many compatibility issues with legacy software. The authors focus very much on rewriting components in OCaml, which would require a lot of work on reimplementation. There are also discussions that OCaml is more secure due to the fact that there are fewer lines of code but this does not correlate with the need to rewrite everything. Furthermore, they claim that because the hypervisor is written in C, it is insecure and should be rewritten in OCaml.

3.2.2 Perfomance Tests

Although in terms of information they generally talk about Unikernels, the authors focus on their MirageOS solution. Most performance tests are done compared to Linux and in some places to MiniOS, another Unikernel. Moreover, every time the performance seems worse than other systems, the authors invoke the type-safety of OCaml, which they say would be a good tradeoff, although this is not necessarily true.

3.2.3 Threading Support

MirageOS has only a cooperative thread scheduler that even the authors present as a downside: "the use of a cooperative threading .. as a single bug can completely deadlock an entire appliance"

3.3 Alternative approaches

Although MirageOS was a novelty in the world of operating systems at the time of writing this paper, there is now a wider range of Unikernels. Probably the most important aspect would be the compatibility with existing software. In order to catch on to the public, Unikernels need to reach a level of maturity where we don't need to patch components like the hypervisor just for them work.

4 *My VM is lighter (and safer) than your Container*

This paper[4] brings up an alternative to the usage of containers, in the form of a lightweight hypervisor, LightVM and a minimalist OS, based on Linux, Tinyx. The authors want to show, as the paper title suggests, that a virtual machine, ran by a hypervisor, can be both smaller, faster and more secure than a container.

4.1 Strengths

4.1.1 Relevant comparison points

One aspect that differentiates this paper from the others is what the authors have decided to have as a comparison point: they don't make the comparison only with large distributions of Linux, but, instead, they center the whole evaluation around already-optimised solutions, like unikernels and Docker containers, getting better performance with their minimalist OS.

4.1.2 Solving the problems

The paper also manages to solve a problem that it identifies: as more VMs are started on a hypervisor, the starting time increases. Their solution provides constant starting time, irrespective of how may VMs are already booted. This is achieved by identifying problems that Xen, the hypervisor on which LightVM is based, solving those problems and adding some innovative features, like the pre-allocation of VMs.

4.1.3 Very good use cases

Although the other strengths of the paper are not to be ignored, we consider that the strongest part of this paper is the one where the use cases are described. Showing not only what the authors have achieved, but giving ideas to potential users on how to best utilise their product sets this paper apart from the others.

4.2 Weaknesses

4.2.1 High complexity

We consider that one of the main weaknesses of this paper is its complexity, and the level of detail in which the authors go. There are two main items that are discussed: LightVM and Tinyx. Each of those two could have a paper on its own. The complexity of the paper arises from the need to describe the internals of Xen, in order to be able to fully describe the implemented solution, which brings another weak point: the solution is very Xen-centric, placing it on a niche of the virtualization domain, as Xen is not easily-accessible, unlike, KVM, Hyper-V, ESXI and other hypervisors.

4.2.2 Tinyx is overshadowed

Having mentioned that the paper could be, perhaps, split in two, another weak point can be found: Tinyx is mentioned as part of the paper (”The development of Tinyx, an automated system for building minimalistic Linux-based VMs”), but details about it are scarce, and it is overshadowed by LightVM. The reader learns only what it is, why it exists, but not how it was done, more in depth.

4.3 Alternative approaches

The idea of having smaller and more efficient hypervisors is an actual one, with Firecracker achieving this same ideal. Also, the idea of minimal VMs is also widely explored today, having projects like Alpine Linux or unikernels, like Unikraft. The current paper, even if not fully implemented, can be considered a ramp for many projects that try to minimize hypervisors and virtual machines.

5 Paper comparison

5.1 Objectives

All three papers have one thing in common: Highly specialized solutions for safer and faster virtualization. Although the subject of unikernels is touched upon in all of them, they have vastly different approaches.

5.1.1 Security

Although, by design, unikernels are more secure than an ordinary virtual machine, The first paper does not even touch upon security as an objective, instead aiming for performance, simplicity, and portability. The other two papers mention security from the abstract, clearly define it as an objective, and explore it as an use case.

The second and the third paper both do a good job at exposing weaknesses of existing solutions, and defining a clear role for their respective implementations.

5.1.2 Performance

All three papers have vastly different implementations despite the fact that performance is one of the primary objectives in all of them.

As far as benchmarking methodology goes, the last paper does the best job to illustrate direct effects of their work, while the other two papers have less than fair comparisons, using stock virtual machines, without any modifications that attempt to make them lighter.

The first two have less than stellar results, with IncludeOS having less-than-expected results as far as overhead, MirageOS not being compared to many other relevant solutions, and when results are not favorable, the second paper's authors justify it as a necessary compromise to achieve other objectives.

With regard to memory overhead and footprint, all three papers showcase productive and useful results.

5.2 Project Scope

The first paper seems to have the most approachable implementation, since it uses C++ as the language of choice. There is a mature toolchain ecosystem, and although developing an OS seems like an insurmountable task, it has been done before, in the case of SerenityOS [5]. Of course, adapting to the new development environment adds a lot of complexity and development cost, but so do the other two papers.

The second paper takes it one step further, using a rather esoteric language. All the weaknesses of the previous paper apply, to which we add the performance penalties the language's abstractions incur, and the sacrifice of compatibility, due to mirageOS not being able to target as many appliances as the first paper. The authors' claim for more software being rewritten in OCaml is also outside the majority's scope, barring for OCaml fans.

The third paper is by far the broadest in terms of scope. They have made ample modifications to core components of the Xen hypervisor. They also present Tinyx as a solution but never break it down within the paper, so there is a whole component of unknown complexity. Even as we presented the solution as being the best in terms of performance and overhead improvements, it is very hard to upstream the work made, due to the sheer maintenance cost of all the patched components. Even if it only targets Xen, it targeted core components and made ample modifications to the point they had to dedicate entire pages to explaining said components.

5.3 Writing Style

It is clear that the authors have different levels of experience in academic writing, and each is the product of their academic environment, each with its advantages and disadvantages.

The first paper has some grammatical errors, but the overall language was not overly complex and inaccessible. The paper shows us a very good separation between subsections.

It is one of the first papers to write about the modern unikernel almost as we know it today, and it certainly has its place as an introduction to unikernels.

The second paper has an even simpler language, not going into too many details. While at times it feels like the authors are simply touting OCaml as the right means to their unikernel end, but it does a very good job at describing the problem, and offering the reader a solid experience as they explore the solution. A moderately better read than the first paper if the reader has never read about unikernels before.

The third paper takes everything to the next level. The language becomes much more technical, there are whole sections dedicated to Xen components, and things are explored in a much more detailed way. It is clear that this paper was made for a much more niche audience, exploring an elusive problem. While now one could make the argument that KVM is of more interest as of today, this paper makes for quintessential learning material for people who want to hack their favourite virtualization platform. It is not unikernel-centric, but since unikernels are so closely tied with virtualization in general, it makes for a wonderful complementary read to the other papers that simply goes a level deeper.

6 Conclusion

To sum it everything up, all three papers have their ups and downs. Some went into too much detail, some did not go into enough detail. What is clear is that each paper approaches the subject of unikernels differently. Some try to implement them, others try to rewrite them, and others concentrate on the hypervisors themselves.

As such, the articles shouldn't be ranked based on their goals and results. The only comparison done is at a paper quality level. In this category, it is safe to say that the latter articles offer a much higher writing quality than the first, even if they maybe go into too much detail.

Still, all three articles explored (at their time) new frontiers and paved the way for the work that goes today into unikernels.

References

[1] Haugerud H. Brattedrud A. Walla A-A. *IncludeOS: A minimal, resource efficient uniker-nel for cloud services.* https://oda.oslomet.no/oda-xmlui/handle/10642/3189.

[2] Charalampos Rotsos Anil Madhavapeddy Richard Mortier. *Unikernels: Library Operating Systems for the Cloud.* https://mort.io/publications/pdf/asplos13-unikernels.pdf.

[3] OCaml Community. *OCaml official website.* https://ocaml.org/.

[4] Filipe Manco, Florian Schmidt, Costin Raiciu, Felipe Huici. *My VM is Lighter (and Safer) than your Container.* https://dl.acm.org/doi/10.1145/3132747.3132763.

[5] Andreas Kling, SerenityOS Community. *SerenityOS official website.* https://serenityos.org/.